VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Martin Kahánek

Bakalářská práce

Vedoucí práce: Ing. Lenka Skanderová, Ph.D.

Ostrava, 2021

Abstrakt

Tato bakalářská práce popisuje mou individuální odbornou praxi na projektu London Theatre Direct ve společnosti ICT Capital s.r.o. První část práce pokrývá společnost ICT Capital a projekt London Theatre Direct, jehož součástí bylo zadání této práce. Druhá část se již zcela zabývá problematikou digitálních peněženek, jejich integrací a samotným zadáním práce, jeho analýzou a implementací. Závěr je věnován celkovému zhodnocení odborné praxe a znalostem využitým, či nabytým během času stráveném na praxi.

Klíčová slova

.NET; API; Apple; Google; digitální lístek; digitální peněženka; E-Wallet

Abstract

This bachelor thesis describes my individual professional practice on the project London Theatre Direct in the company ICT Capital s.r.o. The first part of the thesis covers the ICT Capital company and the London Theatre Direct project, which included the assignment of this work. The second part deals fully with the issue of digital wallets, their integration, and the actual assignment, its analysis and implementation. The conclusion is dedicated to the overall evaluation of professional practice and knowledge used or acquired during the time spent in practice.

Keywords

.NET; API; Apple; Google; digital pass; digital wallet; E-Wallet

Poděkování

Rád bych tímto poděkoval vedoucí mé práce Ing. Lence Skanderové, Ph.D. za nespočet odborných konzultací a rad, bez kterých by tato práce nedosahovala očekávané kvality. Dále bych rád poděkoval Mgr., Ing. Miroslavu Kaděrovi a celému vývojářskému týmu projektu London Theatre Direct za příkladnou spolupráci a vedení během mé praxe ve společnosti ICT Capital s.r.o.

Obsah

Seznam použitých zkratek a symbolů

HTML	–	Hyper Text Markup Language
CSS	–	Cascading Style Sheets
REST	–	Representional State Transfer
API	–	Application Programming Interface
URL	–	Uniform Resource Locator
ORM	–	Object Relational Mapping
MIME	–	Multipurpose Internet Mail Extensions
MVVM	–	Model-View-ViewModel
IoT	–	Internet of Things
CI	–	Continuous Integration
CD	–	Continuos Delivery
LTS	–	Long Term Support
B2B	–	Business to Business
HTTP	–	Hypertext Transfer Protocol
OS	–	Operační Systém
JSON	–	JavaScript Object Notation
JWT	–	JSON Web Token
IoC	–	Inversion of Control
DI	–	Dependency Injection
SOAP	–	Simple Object Access Protocol
ERP	–	Enterprise Resource Planning
CRM	–	Customer Relationship Management

Seznam obrázků

Kapitola 1

Úvod

Obsah této práce pojednává o mé odborné praxi ve společnosti ICT Capital s.r.o. V následujících kapitolách se budu podrobně věnovat nejen zadání bakalářské práce a jeho implementaci, ale také samotné společnosti, u níž jsem praxi vykonával, projektu a týmu do nějž jsem byl začleněn.

Ve společnosti ICT Capital jsem působil již na jaře roku 2020 v rámci odborné praxe 2. ročníku bakalářského studia, a díky této úspěšné předchozí spolupráci vznikl základ pro tuto práci. Během mé praxe jsem pracoval výhradně v týmu vývojářů vyvíjejícím řešení pro jednoho z největších zákazníků společnosti, zahraniční společnost London Theatre Direct. Během mého působení v tomto týmu jsem se setkal se všemi aspekty vývoje softwarových řešení, od podpory pro vyvíjený produkt, přes udržování stávajících řešení, až po analýzu a vývoj zcela nových modulů.

V úvodních kapitolách se zaměřují na firmu a projekt, jehož jsem mohl být, díky této praxi, součástí. Čtenář je uveden do fungování společnosti ICT Capital, jejích aktivit týkajících se vývoje software i mimo něj a projektu London Theatre Direct. V kapitole o projektu se věnuji nejen obecným informacím o projektu a jeho vedení, ale zejména architektuře projektu, jejíž probíhající přestavba úzce souvisí s řešením zadání této bakalářské práce. V následujících kapitolách se již plně věnuji doméně, mnou implementovaného zadání, elektronickým vstupenkám a jejich distribuci. Ve čtvrté kapitole se věnuji teorii a motivaci stojící za digitalizací vstupenek a věnuji zde dvě samostatné podkapitoly nejzásadnějším externím platformám, které nám digitalizaci vstupenek poskytují a zejména jejich integraci do implementovaných systémů. V páté a nejdůležitější kapitole se již věnuji samotnému řešení zadání. Podrobně zde popisuji každou jednotlivou fázi řešení, od analýzy návrhu, přes implementaci, až po poznatky, jenž provázely testování finálního řešení. Závěr této práce věnuji jejímu zhodnocení, zkušenostem a znalostem z bakalářského studia uplatněným během praxe a také znalostem, jež jsem nabyl až během praxe, tudíž mi při prvotní práci se zadáním scházely.

Kapitola 2

O společnosti ICT Capital

Společnost ICT Capital [1] byla založena ve Frýdku-Místku v roce 2011. Za 10 let své existence si vytvořila pevné místo na trhu mezi předními dodavateli informačních systémů pro zákazníky z České republiky i zahraničí. Od roku 2009 společnost ICT Capital využívá pro všechny své projekty značku ITIXO, která ve spojení s rozšířením vývojářských týmů a otevření nové kancelářské pobočky v Ostravě, odstartovala novou éru společnosti.

2.1 Zaměření a technologie

Primárním zaměřením společnosti je vývoj aplikací pro webové a desktopové platformy za využití moderních technologií a inovativního přístupu. Všechna řešení společnosti jsou komplexně zastřešena od analýzy, přes implementaci až po hotové řešení a marketingovou prezentaci produktu.

Mezi nespočet úspěšných projektů společnosti se řadí nejen již hotová řešení ERP, CRM či skladových a e-shopových systémů, ale také rozsáhlý kontinuální vývoj a podpora pro své největší zákazníky, či řada inovativních *IoT* projektů a nadstaveb do světově používaných aplikací.

Vývojáři v rámci vývoje sází primárně na technologie od společnosti Microsoft. Drtivá většina projektů je řešena na platformě .NET [2]. Zákaznicky orientované projekty vyžadující pokročilé uživatelské rozhraní jsou implementovány v moderních a stabilních frontendových frameworcích *Vue.js*, nebo *ReactJS*, zatímco podnikatelské aplikace využívají specializovaných a moderních řešení jako *ASP.NET Core Blazor*, či partnerské *DotVVM* [2.2].

Z pohledu infrastruktury a architektury softwarových řešení vývojáři kladou důraz na dlouhodobou udržitelnost a rozšiřitelnost vyvíjených aplikací. Portfolio společnosti tvoří projekty stavěné na tradičních architektonických návrzích s využitím Microsoft SQL Databází, ale také distribuovaná cloudová řešení s využitím platformy Microsoft Azure a technologií jako Azure Service Bus, či Azure Functions a cloudových datových uložišť v popředí s ElasticSearch dokumentovou databází.

Společnost ICT Capital ovšem netvoří jen vývojáři. V rámci *rebrandingu* na značku ITIXO vznikla odnož ITIXO Creative primárně sídlící ve Frýdku-Místku. Tato část firmy, jak již název

napovídá, se zabývá kreativními aktivitami firmy, zejména na poli marketingu a propagace. Mezi hlavní činnosti patří kompletní audiovizuální prezentace projektů firmy, či pořadatelská a produkční činnost na projektu Update Conference [2.2.2].

2.2 Spolupráce

Společnost ICT Capital spolupracuje s předními technologickými i netechnologickými partnery. Mimo programátorskou a kreativní činnost se společnost angažuje také ve vzdělávacích programech pro vývojáře, ať už formou pořádání vývojářských konferencí, ale také aktivní participací v rámci *Windows User Group* Česká republika, která spojuje českou komunitu .NET vývojářů a uživatelů Microsoft technologií.

2.2.1 Společnost Riganti

Úzkou spolupráci, již od svého zrodu, navázala společnost se společností Riganti s.r.o. [3], sídlící zejména v Praze a Brně. Se společností Riganti probíhá spolupráce na několika významných projektech, v rámci jejichž vývoje aktivně využívá vývojářských produktů a knihoven společnosti Riganti, jako například knihovnu *Riganti Utils Infrastructure*.

V rámci partnerství se společností Riganti se vývojáři společnosti ICT Capital podíleli na vývoji frontendového frameworku DotVVM pro podnikatelské aplikace. Tento *open-source* framework, určený pro platofrmu ASP.NET, umožňuje vývojářům vytvářet webové aplikace s pokročilým uživatelským rozhraním za využití jazyka C#. Tento framework aplikuje principy návrhového vzoru *MVVM*, který umožňuje efektivně oddělit prezentační vrstvu a její logiku od business vrstvy aplikace.

2.2.2 Update Conference

Dalším ze stěžejních projektů v rámci partnerství se společností Riganti je zrod projektu Update Conference [4]. Cílem tohoto projektu je pořádání vývojářských konferencí se zaměřením na .NET vývojáře.

Pod hlavičkou značky Update Conference se uskutečnilo již několik menších konferencí ze série Update Days, jenž pokrývají nové technologie a trendy ve světě .NET a jsou určené, až na vyjímky, zejména pro českou komunitu vývojářů. Středobodem celého projektu je ovšem mezinárodní konference Update Conference Prague, která se pyšní titulem největší konference pro .NET vývojáře ve střední Evropě. Update Conference Prague se uskutečnily již 3 ročníky, z toho poslední ročník proběhl, kvůli pandemii koronaviru v roce 2020, jako online událost pod značkou Update Now.

Kapitola 3

Projekt London Theatre Direct

3.1 O projektu

Projekt, jenž je předmětem mé bakalářské práce, je součástí dlouhodobého projektu prodejního a distribučního systému divadelních vstupenek pro společnost London Theatre Direct (dále jen LTD). Společnost LTD je jedním z největších zákazníků společnosti ICT Capital a historie této spolupráce sahá až k samotnému vzniku společnosti ICT Capital.

Doménou společnosti LTD je prodej a distribuce vstupenek, zejména na divadelní představení v Londýně a v řadě dalších velkých britských městech. Pole působnosti společnosti se lineárně zvětšuje a od svého vzniku v roce 1999 společnost již expandovala nejen do jiných zemí Evropy a na New Yorský Broadway, ale také v rámci nabízených služeb. V současnosti portfolio společnosti obsahuje nejen divadelní představení, ale také kina, atrakce, či sportovní události.

3.2 Projektové řízení

Ve společnosti ICT Capital je pro tento projekt vyhrazen celý vývojový tým, specializující se čistě na projekt LTD. Po dobu mé praxe ve firmě jsem měl možnost spolupracovat se všemi kolegy z tohoto týmu. Celý tým zahrnuje *frontend* vývojáře, kolegy zajišťující podporu a zkušený tým *backend* vývojářů, jehož jsem byl výhradně součástí také já na pozici .NET vývojáře.

Pro řízení práce na projektu se využívá sada nástrojů Azure DevOps od společnosti Microsoft. Tato sada cloudových služeb poskytuje vývojovým týmům nejen správu pracovního procesu v podobě např. podpory rozdělení práce do iterativních celků (tzv. sprintů), dle metodiky agilního vývoje *SCRUM*, ale také podpůrné nástroje pro samotný vývoj, jako verzovací nástroj *Git*, či podpora kontinuálních praktik integrace a nasazení aplikací, *CI/CD*.

3.3 Technologická řešení a architektura

Ač se jedná o rozsáhlý a komplexní projekt, technologická stránka projektu je poměrně konzistentní. Veškeré infrastrukturní aplikace a jádro projektu jsou vyvíjeny na platformě .NET, konkrétně v současné době paralelně běží aplikace postavené na .NET Framework 4.x (zejména tzv. *legacy* aplikace, tedy zastaralá řešení) a zároveň modernizované, či úplně nové aplikace postavené na platformě *.NET Core 3.1*, která je zároveň aktuální *LTS* verzí. S postupnou modernizací do projektu proniká i integrace cloudových služeb z platformy *Azure*.

3.3.1 Struktura projektu

Jak již bylo řečeno v úvodu této podkapitoly, jedná se o velmi rozsáhlý projekt, který je rozdělen do několika úzce svázaných částí, zajišťujících nejen bezproblémový nákup vstupenek zákazníkem, ale také celkovou správu systému či partnerská řešení.

Webový portál LondonTheatreDirect.com

Prodejní webový portál je středobodem celého projektu. V době mé praxe se tento web nacházel v tranzitivní fázi, přechodu z implementace postavené na *ASP.NET WebForms* na novou implementaci na platformě *ASP.NET Core Razor Pages* a *ReactJS*.

Webový portál poskytuje služby od běžného listování představeními, přes samotnou rezervaci vstupenek skrze interaktivní výběr sedadel, až po dokončení objednávky s řadou integrovaných platebních metod jako *Apple Pay*, *Amazon Pay*, či *Google Pay*.

Administrace

Administrační webová aplikace slouží výhradně pro správu celého prodejního systému. Skrze tuto *ASP.NET WebForms* aplikaci lze nejen spravovat obsah webového portálu, ale zejména je zde implementováno kontrolní centrum nad všemi rezervacemi a transakcemi protékajícími prodejním systémem.

Tato aplikace není veřejná a je určena pouze pro vývojový tým nebo zaměstnance společnosti LTD díky implementované autentizaci s dvoufaktorovým ověřením.

Partnerská řešení

Partnerská řešení poskytovaná společností LTD jsou postavena nejen na vystavených *REST* a *SOAP API*, ale také pomocí nabízení e-shop portálu pod personalizovaným designem daného partnera.

Speciální sekcí je také *B2B* řešení, které funguje jako separátní webová aplikace zprostředkovávající obchodní styk mezi společnostmi na principu *business to business*.

Infrastrukturní aplikace

Infrastrukturní aplikace jsou nezbytnou součástí prodejního systému. Tato neveřejná síť podpůrných aplikací starajících se o synchronizaci dat, či zajišťujících konektivitu s dodavatelskými systémy většinou běží na bázi služeb a v pravidelných intervalech provádí jim přidělenou úlohu.

Komunikace mezi těmito aplikacemi je zajištěna na bázi zasílání a čtení zpráv z tzv. fronty skrze službu *Azure Service Bus*. Aplikace si z fronty odebírá jí určené zprávy a na základě jejich obsahu je zpracovává, nebo vyvolává určitou akci, např. zmíněnou synchronizaci dat.

3.3.2 Architektura

Původní řešení celého projektu bylo založeno na monolitické architektuře, která v současné době technologických možností byla jen velice obtížně rozšiřitelná. Na základě požadavku zákazníka a následné analýzy, jíž jsem se účastnil v rámci praxe ve 2. ročníku bakalářského studia, byl tento monolitický celek rozčleněn do několika oddělených mikroslužeb na základě jejich oddělených domén, které by tyto služby měly řešit.

Monolit

Monolitická aplikace [5] se vyznačuje tzv. *all-in-one* (česky *vše v jednom*) řešením ve vývoji software. V monolitu je drtivá většina, ne-li veškerá, aplikační, business a datová logika umístěna v několika vrstvách a knihovnách v rámci jednoho projektu.

Aplikace vyvíjené monoliticky jsou zpravidla vyvíjeny nad jednou platformou a databází. Z hlediska rozdělení samotného kódu se uplatňuje většinou třívrstvá architektura [6], která ovšem musí být striktně dodržována z důvodu udržitelnosti kódu.

Komunikace mezi vrstvami a částmi kódu v monolitických aplikacích je založena na volání metod jednotlivých objektů z různých vrstev. Tento způsob se vyznačuje svou rychlostí a jednoduchostí, protože do samotné komunikace nevstupuje žádné zpoždění, které přináší např. *HTTP* komunikace.

V současné době se od monolitických aplikací spíše upouští, i když určitě mají své nesporné výhody, mezi které patří např. rychlost zaučení nového vývojaře, či transparentní ladění celé aplikace. Nevýhody tvoří spíše technologické aspekty, v popředí s omezenou škálovatelností, kdy pro škálování kritické části systému je nutné přidat instanci celé aplikaci namísto dané části, nebo postupem času komplikovanější rozšiřitelnost. Tyto nevýhody mohou vývoj takto stavěné aplikace znepříjemnit, či dokonce dále zcela znemožnit.

Mikroslužba

U mikroslužeb [7] nemusí být pevně daná jedna platforma, databáze, či dokonce architektura jednotlivých služeb, právě naopak. Každá jedna služba by měla být v rámci vývoje vlastněna (anglicky *owned*), jedním vývojovým týmem, či vývojářem. Samotný tým nebo vývojář vývojář by potom měl

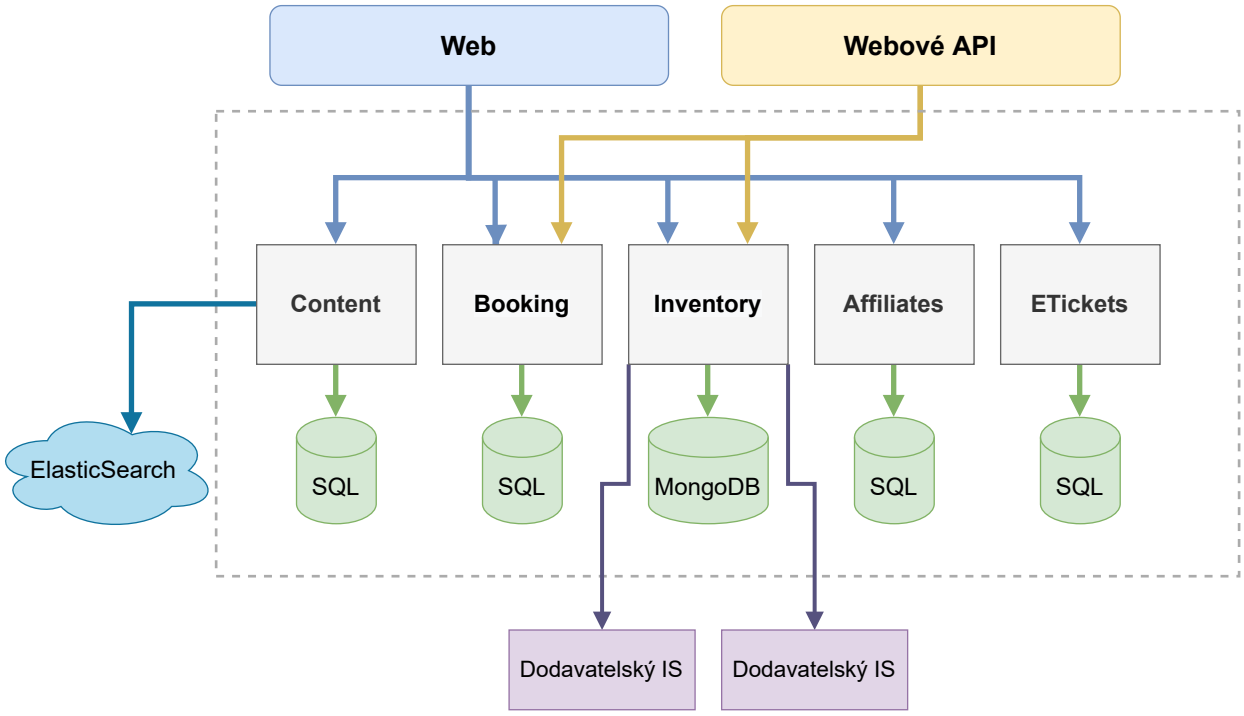
určit, která platforma, datové uložiště, či architektura bude ideální pro doménu, kterou má daná mikroslužba zastřešit.

Jednotlivé mikroslužby by měly být tzv. *loosely-coupled*, což v překladu znamená volně vázány. V případě úzké vazby mezi službami by takové služby ve většině případů měly být sloučeny do jedné. Samotné domény jednotlivých služeb, jak již bylo řečeno, by měly být co nejvíce nezávislé.

Komunikace mezi těmito celky je založena na řadě komunikačních způsobů, mezi nejpoužívanější se řadí *REST*, *gRPC*, nebo jakýkoliv princip *messagingu*, tedy zprávách, kdy jedna služba zprávu publikuje do tzv. fronty a druhá služba, jenž tyto zprávy odebírá, je dále zpracovává.

Z hlediska rozšiřitelnosti, škálování a udržitelnosti se nabízí hned několik výhod. Pro úpravu, či rozšíření doménové logiky je třeba upravit a nasadit do produkce jen službu, která danou logiku řeší. Pokud je daná logika kritickou pro funkčnost celé aplikace je možné danou službu nasadit ve více instancích a tím pokrýt případný nápor na danou část aplikace.

Za speciální stav ve vývoji mikroslužeb lze považovat tzv. distribuovaný monolit [8]. Tento jakýsi mezistupeň v přechodu na mikroslužby je již sám rozdělen na jednotlivé služby. Tyto služby bývají zpravidla úzce spjaty (anglicky *tightly-coupeled*) nějakým pojícím prvkem, ať už silnou datovou vrstvou, nebo hustou komunikací mezi službami. Distribuovaný monolit se často považuje za tzv. *anti-pattern* (česky *antivzor*). Je ovšem nespornou součástí modernizace a přechodu historických systémů na mikroslužby. V době vzniku této práce se projekt London Theatre Direct nachází právě ve fázi rozdělování datové vrstvy a odstraňování závislostí jednotlivých služeb na sdílené databázi.



Obrázek 3.1: Schéma mikroslužeb pro prodejní systém projektu LTD

Kapitola 4

E-Wallet a digitalizace vstupenek

Již dlouhou dobu žijeme v tzv. digitální éře, kdy trend digitalizace pohlcuje všechny aspekty běžného života. Tyká se to také obyčejných peněženek. S příchodem platebních karet přišla také revoluce v placení za služby a produkty. Zákazník jednoduše vloží, nebo přiloží platební kartu k terminálu a bezhotovostní transakce je provedena.

Digitalizace za poslední roky postoupila do takové fáze, že většinou již nepotřebujeme ani zmíněné karetní řešení, ale s nástupem chytrých telefonů nám postačí právě jedno takové zařízení obsahující elektronickou, nebo chcete-li digitální peněženku tzv. E-Wallet.

4.1 E-Wallet

Pojmem E-Wallet, nebo-li elektronická peněženka, můžeme chápat jakoukoliv softwarovou aplikaci, nebo elektronické zařízení jehož účelem je zprostředkování peněžních a kupónových transakcí skrze uživatelem nahranou platební kartu, či kupón. Nemusí ovšem jít jen o jedno zařízení, nebo aplikaci, ale z drtivé většiny jde již o komplexní systém zajišťující tuto službu.

Nejvíce perspektivní platformou pro elektronické peněženky jsou právě mobilní zařízení. V této době těmto zařízením zcela dominují platformy, *iOS* od společnosti Apple a *Android* společnosti Google. Obě tyto platformy disponují nativními implementacemi elektronických peněženek.

4.2 Apple Wallet

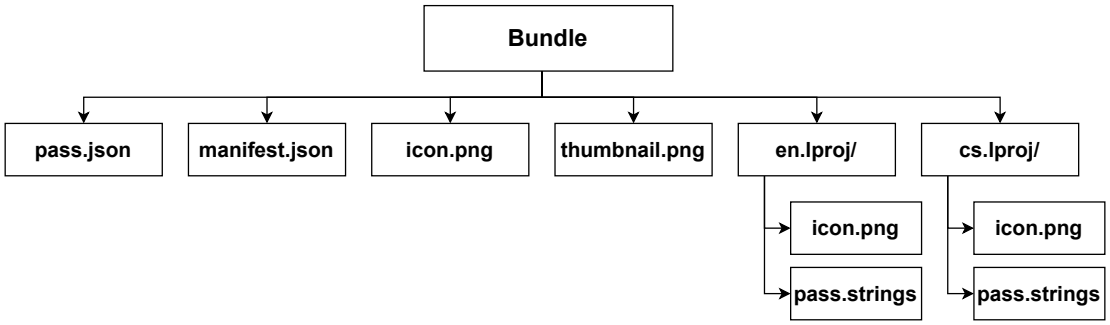
Apple Wallet poskytuje nativní řešení elektronické distribuce digitálních lístků pro platformu *iOS*. Tato služba agreguje kupónové a platební služby, díky kterým lze nejen uplatňovat vstupenky na události, ale také bezkontaktně platit skrze platební terminál.

Generování lístků pro *Apple Wallet* není na straně společnosti Apple řešeno jako poskytnutá služba. Celé řešení musí být implementováno jako *in-house*, nebo-li interní na straně distributora vstupenek, služba podle daného předpisu od společnosti Apple [9].

In-house řešení přináší řadu výhod, kdy jako distributor lístků máme kontrolu nad vygenerovanými lístky a nevzniká tak závislost na externích službách, ale naopak i nevýhody spojené s obsluhou samotných procesů generování a aktualizace lístku, nebo datového uložiště lístků.

4.2.1 Generování lístku

Pro vygenerování lístku pro *Apple Wallet* musíme vytvořit souborovou strukturu, tzv. *bundle*, splňující předepsanou strukturu dle dokumentace služby:



Obrázek 4.1: Struktura souborů adresáře pro generování lístků pro *Apple Wallet* [9]

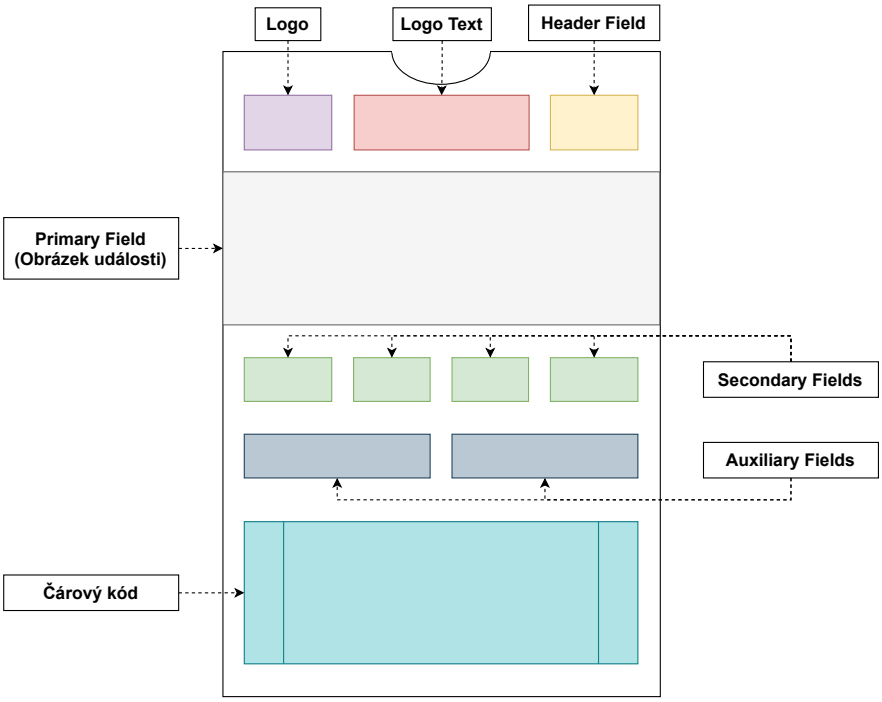
Tento *bundle* obsahuje grafiku, jenž se na lístku zobrazuje, zpravidla logo a poutací obrázek k události. Dále pak obsahuje jednotlivé lokalizace lístku nadefinované při generování a středobodem celého adresáře je soubor *pass.json*, který definuje samotnou strukturu lístku.

4.2.2 Design lístku

Lístek je definován souborem *pass.json* v rámci vygenerovaného *bundle* vstupenky. Apple poskytuje různé formáty lístků, dle jejich využití:

- **Event ticket** - lístek na jednorázovou událost,
- **Boarding pass** - lístek pro dopravní přepravu,
- **Coupon** - jednorázový, např. slevový, kupón,
- **Generic** - obecně využitelný lístek,
- **Store card** - věrnostní karta.

Každá z šablon má svůj předepsaný vzhled a cílový účel, který však není nutno explicitně dodržovat.



Obrázek 4.2: Vzhled šablony *event ticket*

Samotný soubor *pass.json* specifikující lístek obsahuje pět identifikačních klíčů:

- `description` - popis daného lístku, v našem případě název, místo a datum divadelního představení,
- `formatVersion` - verze lístku s výchozí a jedinou možnou hodnotou *1*,
- `passTypeIdentifier` - sdílený identifikátor lístků dané pro danou šablonu,
  - přiřazen společností Apple distributorovi lístků,
  - identifikátor je generován dle konvence pass.com.*<distributorem definovaný název>*.*<název šablony>* [1],
  - identifikátor musí korespondovat s přiřazeným podepisovacím certifikátem,
- `serialNumber` - distributorem určený unikátní identifikátor daného lístku,
- `eventTicket` - klíč definující vybranou šablonu pro daný lístek, v našem případě `eventTicket`.

[1] Název šablony není povinný, specifikuje se pokud distribuujeme různé lístky implementující různé šablony. V našem případě tudíž postačil identifikátor `pass.com.londontheatredirect`, protože specifikujeme lístky jen pomocí *event ticket* šablony

Lístek je rozdělen do jednotlivých polí, jejichž počet a rozložení na lístku si definuje vybraná šablona. Příklad rozložení lze vidět na obrázku 4.2. Samotných typů polí definujeme pět, z toho dva typy jsou velice podobné a rozlišeny jsou pouze z důvodu umístění na lístku.

1. **Header fields** - hlavičková pole mohou být až 3 a zpravidla jedno obsahuje logo distributora,
2. **Primary fields** - obecně obsahující primární informace o lístku. Šablona *event ticket* obsahuje jedno takové pole a je určeno pro poutací grafiku k dané události. Jejich umístění je nejvíce prominentní,
3. **Secondary a Auxiliary fields** - pro obecné informace, zobrazené na přední straně lístku,
4. **Back fields** - pole pro nejméně důležité, či různé obsáhlejší informace. Umístění na zadní straně lístku.

Každému poli definujeme unikátní klíč, zpravidla korespondující s hodnotou daného pole. Dále definujeme hodnotu, nadpis pole, můžeme specifikovat datový typ dat v poli a jejich formátování. Speciální hodnotou, kterou můžeme specifikovat danému poli, je tzv. `changeMessage`, tedy zpráva notifikace, jenž se zobrazí zákazníkovi po úspěšné aktualizaci lístku. Nejdůležitějším polem je pak čárový kód. Tomuto poli specifikujeme typ čárového kódu a hodnotu. Tímto čárovým kódem se dále zákazník prokazuje při kontrole lístku.

Kromě rozložení dat po lístku definujeme také obecný vzhled lístku, kdy můžeme aplikovat personalizaci distributora. Kromě již zmíněného loga v hlavičce lístku, definujeme také barvu pozadí a textu, jde ovšem jen o vzhled titulní strany lístku, zadní strana je již plně v režii aplikace *Apple Wallet*.

Posledním krokem při specifikaci lístku je jeho konfigurace. Součástí této konfigurace pro náš případ je nastavení data relevance lístku a data expirace lístku. Tento časový interval vyznačuje od a do kdy je daný lístek uplatnitelný, tzn. lístek, i když platný, nebude uplatnitelný před začátkem, nebo po konci tohoto intervalu. Zajímavou možností v nastavení je také možnost specifikovat relevantní lokaci. Pokud je tato lokace spolu se vzdáleností nastavena, tak se zákazníkovi při příchodu na dané souřadnice lístek sám nabídne pomocí nativní notifikace daného OS.

4.2.3 Komunikace s webovou službou

Asynchronní komunikace s již vygenerovaným lístkem je jedním z největších benefitů digitální distribuce lístků. Tato komunikace je využívána pro aktualizaci již vygenerovaných lístků a jejich dálkovou správu ze strany distributora. Pro navázání této komunikace mezi zařízením, na němž je lístek uložen, a webovou službou je nutné lístek pro tuto komunikaci nakonfigurovat a na straně webové služby vystavit předepsané *REST API*.

Konfigurace lístku pro komunikaci se serverem distributora

Na straně samotného lístku je nutné specifikovat *url* adresu služby, na které je dané *REST API* dostupné, a se kterým bude zařízení komunikovat. Dále je nutné specifikovat autentizační token, kterým se při komunikaci zařízení se serverem bude autentizovat.

Princip komunikace

Komunikace mezi zařízením a webovou službou započíná již během přidání lístku do aplikace *Apple Wallet*. V rámci tohoto procesu je za využití autentizačního tokenu lístku provedena registrace zařízení k lístku.

Samotný aktualizační proces probíhá na základě jednostranné inicializace. Vyvolat tento proces může buď sám zákazník, když si vyžádá aktualizaci lístku, nebo distributor na základě tzv. *push* notifikace. Na základě jednoho ze zmíněných podnětů provede zařízení sekvenci volání na webovou službu, dostupné na adrese specifikované v rámci konfigurace lístku.

Push notifikace

Push notifikace jsou součástí systému vzdálené komunikace mezi serverem a zařízením, které tyto notifikace podporuje. Díky tomuto systému můžeme uživateli zaslat a zobrazit notifikaci na mobilním zařízení v případě, že je třeba uživatele o něčem informovat.

Obecně tato komunikace probíhá na základě inicializace ze strany aplikačního serveru, který naváže komunikaci se servery platformy, na niž běží OS daného zařízení, které zadanou zprávu zašlou určeným zařízením a zařízení na základě obdržené zprávy zobrazí notifikační zprávu.

Při práci s elektronickými lístky pro Apple zařízení je tato komunikace stěžejní v procesu vzdálené aktualizace lístku. Komunikace mezi distributorskými servery a zařízením zákazníka je vedena skrze službu *APNs* (Apple Push Notification service). Po úspěšné aktualizaci je tato notifikace zaslána na základě *push* tokenu, přiřazenému zařízení zákazníka, zákazníkovi, jehož zařízení si na základě této notifikace stáhne novou verzi lístku a zobrazí zprávu o upravených polích specifikovanou jako `changeMessage` daných polí.

Webová služba

Webová služba implementovaná na straně distributora je založena na komunikaci skrze vystavené *REST API* a musí poskytovat předepsané rozhraní. Toto rozhraní poskytuje sadu koncových bodů zajišťujících persistenci a operace nad daty jednotlivých lístků a zařízení.

Registrace k vygenerovanému lístku je prvním úkonem pro zařízení v rámci komunikace s webovou službou [4.2.3]. Tento proces probíhá na základě *HTTP Post* požadavku na podadresu:

/<version>/devices/<deviceLibraryIdentifier>/registrations/<passTypeIdentifier>/<serialNo>

Jelikož se jedná o *HTTP Post* požadavek, tak v těle požadavku je zaslán `pushToken`, jenž byl přiřazen danému zařízení při navázání komunikace s *APNs*. Na základě této hodnoty a parametrů `deviceLibraryIdentifier` (identifikátor zařízení), `passTypeIdentifier` a `serialNumber`, z cesty požadavku, je vytvořena registrace zařízení k zadanému lístku.

Načtení sériových kódů lístků pro dané zařízení je vyvoláno v rámci aktualizace lístku skrze push notifikaci, nebo při úspěšné registraci zařízení k lístku. V tomto případě se jedná o *HTTP Get* požadavek, který je směřován na následující podadresu:

/<version>/devices/<deviceLibraryIdentifier>/registrations/<passTypeIdentifier>

Součástí tohoto požadavku může být také nepovinný parametr `passesUpdatedSince`. Na základě tohoto parametru je zde implementována jednoduchá rozhodovací logika, která pro poskytnutou časovou známku vrátí jen ta sériová čísla lístků, která byla od poskytnutého data upravena.

```
{
  "serialNumbers": [ "02928", "0938", "02983" ],
  "lastUpdated": "1351981923"
}
```

Výpis kódu 4.1: JSON formát úspěšného volání koncového bodu `GetSerialNumbers`

Načtení poslední verze lístku probíhá na základě úspěšného načtení kolekce sériových čísel lístků, na něž je zařízení zaregistrováno. Tento *HTTP GET* požadavek je vyvolán automaticky pro každý lístek v kolekci a je směřován na podadresu:

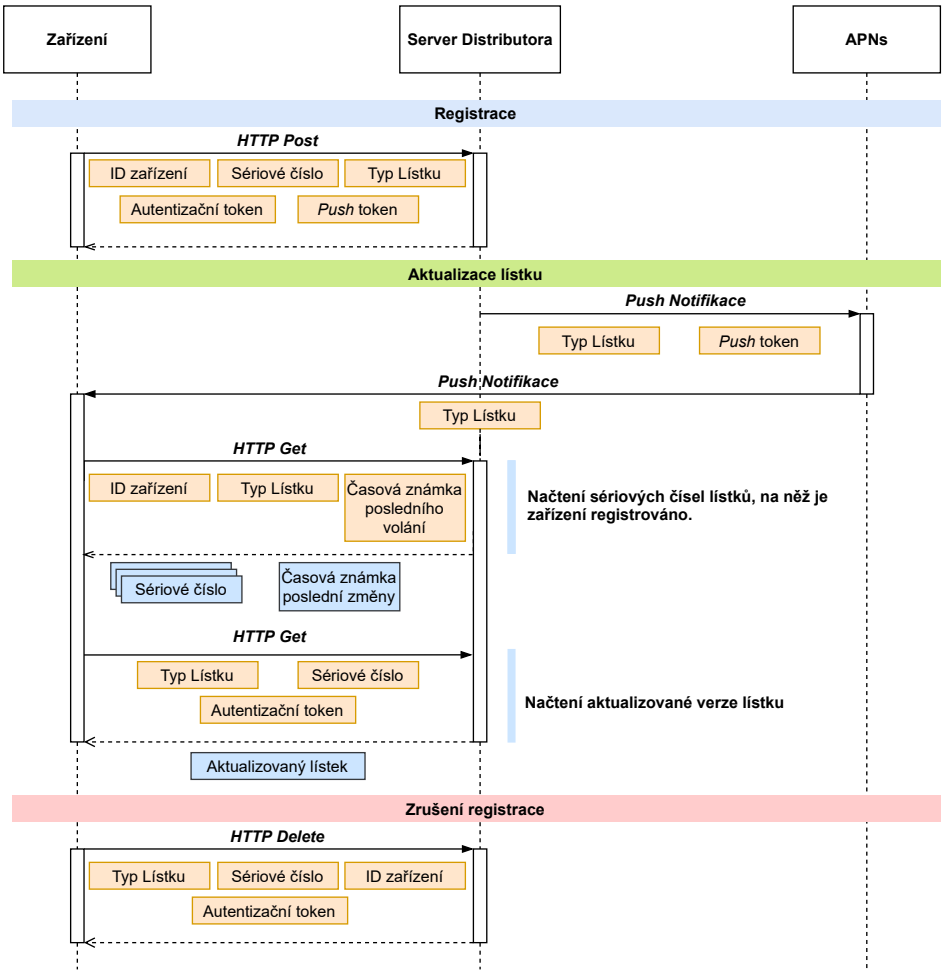
/<version>/passes/<passTypeIdentifier>/<serialNo>

Tento požadavek využívá *HTTP* mechanismus *if modified since*, který funguje na principu porovnání *if-modified-since* hlavičky požadavku, obsahující časovou stopu z výsledku předchozího volání s datem poslední úpravy lístku. Na základě tohoto porovnání vrátí buď poslední verzi lístku, nebo stavový kód 304, nezměněno.

Logování je využito pokud při komunikaci dojde k neočekávaným chybám, které naruší stanovenou sekvenci volání. Tyto chybové hlášky, zachycené zařízením, jsou pak zasílány, jako *HTTP Post* požadavek na koncový bod na podadrese:

/<version>/log

Zrušení registrace ukončuje komunikační vztah mezi zařízením a serverem distributora. Registrace je zrušena smazáním lístku ze zařízení, na základě této události je zavolána stejná podadresa jako při registraci (viz. strana 19), tentokrát však jako *HTTP Delete* požadavek. V rámci tohoto požadavku je buď zrušena samotná registrace zařízení k lístku, nebo pokud se jedná o poslední registraci zařízení v systému, tak je ze systému smazáno i zařízení.



Obrázek 4.3: Komunikace mezi zařízením a serverem v rámci integrace Apple

4.2.4 Distribuce lístků

Distribuce lístků probíhá na principu *on-demand*, čili na vyžádání. Řádně vygenerovaný lístek je podepsán dvěma certifikáty. Prvním z nich je *Apple Worldwide Developer Relations Intermediate Certificate* (zkráceně *WWDR* certifikát). Druhým je pak tzv. *Pass Type Id Certificate*.

Pass Type Id certifikát obsahuje klíče přiřazené distributorovi. Naopak *WWDR* certifikát obsahuje veřejné klíče k certifikační autoritě společnosti Apple a slouží tedy jako jakýsi most mezi distributorským certifikátem a certifikační autoritou společnosti Apple.

Samotný lístek je podepsán pomocí tzv. `manifest.json` souboru, který obsahuje slovník relativních cest k souborům v rámci adresářové struktury lístku a zakódovaného obsahu daných souborů s časovou známkou jeho podpisu.

Podepsaný lístek je zabalen do ZIP archivu, který je, za využití *MIME* typu přenášeného média *application/vnd.apple.pkpass*, distribuován zákazníkům. Je nutno podotknout, že takto stažený lístek je možno konzumovat jen z operačních systémů *iOS* a *MacOS*.

4.3 Google Pay

Google Pay je dnes výhradní implementací elektronické peněženek pro mobilní zařízení s OS Android a díky rozsahu služeb společnosti Google je její přesah patrný i v běžném internetovém světě. Účet k této službě je totiž dostupný z jakéhokoliv zařízení s přístupem k internetu a přidruženému Google účtu.

Na rozdíl od integrace lístků pro aplikaci *Apple Wallet*, poskytuje společnost Google vlastní webové *API* řešení pro generování a distribuci lístků. Veškeré lístky jsou dále drženy v rámci přiřazeného *Google Pay API for Passes Merchant Centre* účtu, kterým je distributor lístků registrován pro distribuci lístků.

4.3.1 Google Pay API for Passes

Stěžejní službou pro distribuci lístků pro aplikaci *Google Pay* je služba *Google Pay API for Passes* [10]. Tato služba, jak již její název napovídá, je webové *REST API*, skrze které proudí veškerá komunikace mezi zařízením uživatele a servery distributora lístků.

V rámci této služby probíhá jak zmíněná registrace distributora, tak registrace aplikace skrze kterou lístky budou distribuovány.

4.3.2 Generování lístku

Každý lístek se skládá z třídy lístků a objektu lístku. Tyto entity a jejich obsah určují nejen vzhled a korektní konfiguraci lístku, ale také hierarchickou strukturu v námi vygenerovaných lístcích. Třída je vygenerována jako první a slouží jako jakýsi předpis pro objekty. Třída zastřešuje provázanou skupinu lístků, a udává jim společné informace, např. informace o letu. Objekty jsou pak samotné letenky, odkazující na daná sedadla letu specifikovaného třídou lístků.

4.3.3 Design lístku

Podobně jako u lístků pro *Apple Wallet* [4.2.2] jsou lístky pro *Google Pay* rozděleny do několika šablon, které jako distributor můžeme využít:

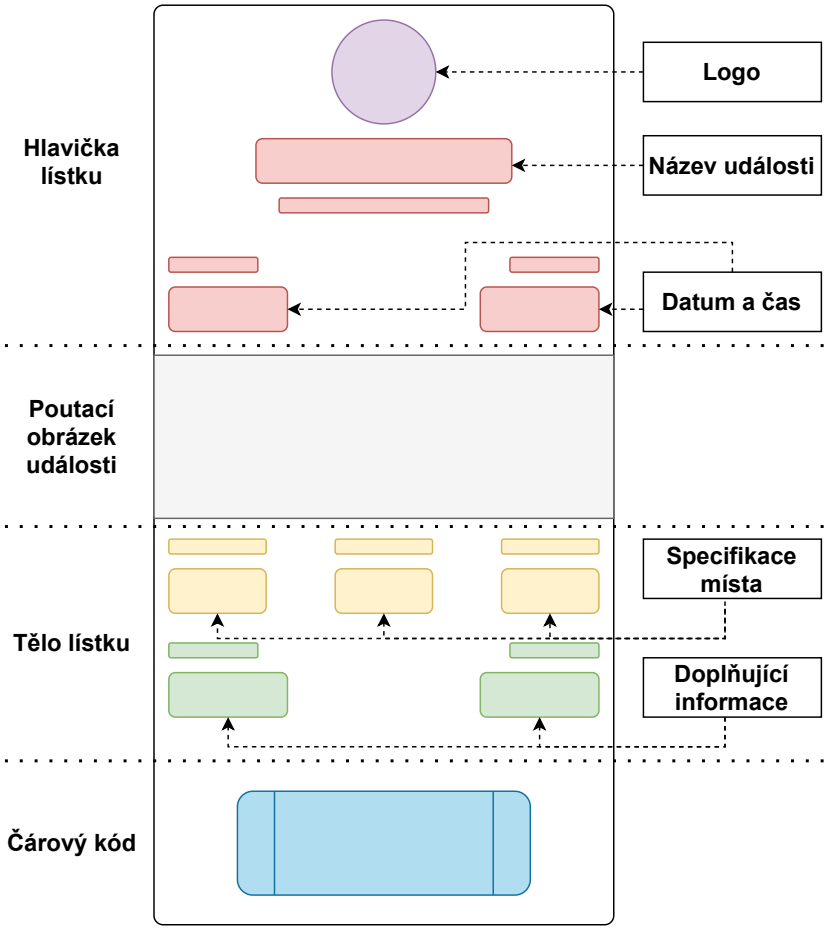
- **Event ticket** - lístek na jednorázovou událost,
- **Boarding pass** - palubní lístky,
- **Gift cards** - dárkový poukaz,
- **Loyalty** - věrnostní karta,
- **Offers** - slevový kupón,
- **Transit passes** - průkaz na hromadnou dopravu.

Stejně jako u Apple lístků [4.2] se tyto šablony nedají vzhledově změnit. Vzhledem k zaměření této práce se opět během následujících odstavců budu věnovat výhradně šabloně *event ticket*.

Z hlediska designu je stěžejní zejména třída lístků a její vlastnosti `logo, hexBackgroundColor` a `images`, které nejvýrazněji definují vzhled lístku. Z datového pohledu je důležitější již objekt lístku, který v sobě nese veškeré identifikátory dané instance lístku a zejména čárový kód pro jeho úspěšné uplatnění.

Lístek samotný je rozdělen čtyř sekcí:

1. **Hlavička** - tvoří ji zejména data specifikované již v třídě lístků (logo, název, atd.),
2. **Tělo** - data specifikující daný objekt třídy (sedadlo, řada, atd.),
3. **Čárový kód** - je oddělen od zbytku obsahu, aby byla zaručena dobrá snímatelnost čtečkou čárových kódů,
4. **Detaily** - obsahují doplňující informace k lístku, zpravidla na zadní straně.



Obrázek 4.4: Rozložení polí na *Google Pay* lístku pro šablonu *event ticket*

4.3.4 Konfigurace lístku

Obě dvě entity tvořící lístek, tedy třída a objekt lístku, jsou v rámci možností konfigurovatelné. V rámci třídy lze definovat lokaci, kde jsou lístky uplatnitelné a dále popisky jednotlivých polí na lístku. Objektu lístku lze naopak konfigurovat časový interval, v kterém je lístek uplatnitelný.

Společné prvky konfigurace pro obě entity pak tvoří zprávy uživateli, které přidáváme do entit při úpravě entity. Tyto zprávy se pak uživateli zobrazí po úspěšném načtení nové verze lístku.

4.3.5 Komunikace skrze Google Pay API for Passes

Veškeré url adresy, které budu zmiňovat v příštích odstavcích se v cestě liší specifikátorem, který určuje o kterou entitu se jedná, zda o třídu, nebo objekt. Pro zjednodušení na místě této specifikace využiji zástupný symbol *<object|class>*.

Komunikace mezi serverem distributora a *Google Pay API* je autentizována skrze *bearer* token v hlavičce každého požadavku a službu *Google OAuth 2.0*, která tyto tokeny ověřuje.

Načtení objektu/třídy ve formátu *JSON* probíhá dle zadaného identifikátoru pomocí *HTTP Get* požadavku na url adresu:

https://walletobjects.googleapis.com/walletobjects/v1/<object|class>/<identifikátor>

Načtení seznamu všech objektů/tříd pro daný typ třídy/objektu, za pomocí *HTTP Get* požadavku na url adresu:

https://walletobjects.googleapis.com/walletobjects/v1/<object|class>

Přidání nového objektu/třídy probíhá po vytvoření lístku na straně distributora. V rámci těla *HTTP Post* požadavku je dále poslán zadaný objekt ve formátu *JSON* na url adresu:

https://walletobjects.googleapis.com/walletobjects/v1/<object|class>

Zaslání zprávy zákazníkovi je jediný čistě specifický případ v rámci této komunikace. Zprávy zákazníkovi můžeme buď přidávat přímo k objektu, či třídě, nebo je zaslat skrze *HTTP Post* požadavek na url adresu:

https://walletobjects.googleapis.com/walletobjects/v1/<object|class>/<identifikátor>/addMessage

V rámci těla requestu je zaslána instance objektu `AddMessageRequest`. Odpovědí je pak upravená verze objektu lístku, či třídy lístků, obsahující přidanou zprávu.

Úprava objektu/třídy je řešena dvěma způsoby, a to *HTTP Patch* a *HTTP Put* požadavky. Oba vyžadují v těle požadavku instanci dané entity. Tento požadavek je směřován na url adresu:

https://walletobjects.googleapis.com/walletobjects/v1/<object|class>/<identifikátor>

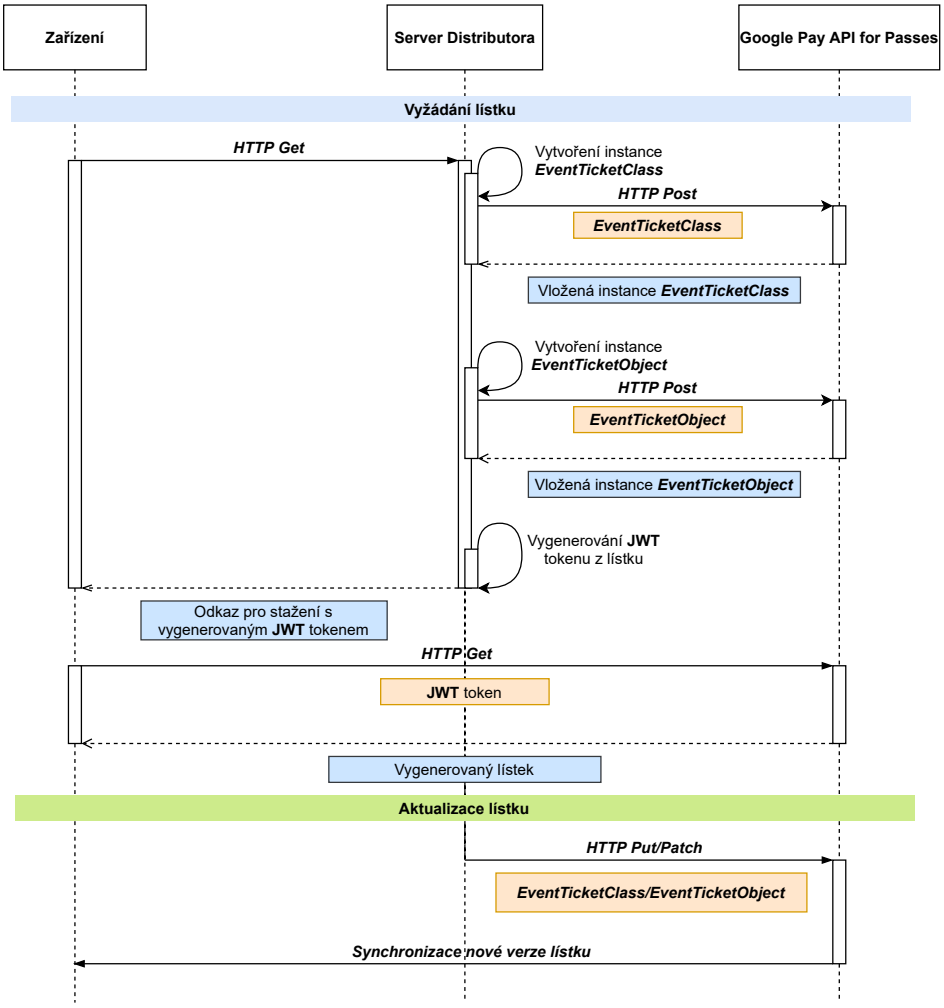
Obě tyto metody v rámci odpovědi vrací upravenou verzi entity. *Put* požadavek na základě identifikátoru zamění entitu za entitu odeslanou v těle požadavku, na druhé straně *Patch* požadavek na základě identifikátoru porovná původní a novou verzi entity a upraví pouze změněné hodnoty.

4.3.6 Distribuce lístků

Stejně jako u *Apple Wallet* lístků [4.2.4], probíhá distribuce a generování lístků pro *Google pay* na principu *on-demand*. Vygenerovaný lístek není poskytován přímo aplikací distributora, nýbrž je na základě identifikátorů třídy a objektu lístku sestaven tzv. *JWT* token. Tento token je součástí url adresy na kterou je zákazník přesměrován a lístek nabídnut k uložení.

Url adresa ke stažení lístku je ve tvaru:

https://pay.google.com/gp/v/save/<JWTToken>



Obrázek 4.5: Komunikace mezi zařízením, servery distributora a Google API

Kapitola 5

London Theatre Direct ETickets

Zadáním této bakalářské práce byla implementace jedné z mikroslužeb, tvořící systém pro prodej a distribuci vstupenek v rámci projektu London Theatre Direct. Doménu této služby tvořil závěrečný proces při nákupu lístků zákazníkem, tedy samotná distribuce lístků v elektronické podobě. Jako řešení byly vybrány digitální lístky pro mobilní zařízení, zejména tedy pro aplikace *Apple Wallet* [4.2] a *Google Pay* [4.3].

Separátní modul, zastřešující tuto část procesu prodeje vstupenek na divadelní představení, byl součástí rozsáhlé přestavby celého systému a vzhledem k faktu, že digitální lístky pro mobilní zařízení jsou v rámci projektu London Theatre Direct [3.1] zcela nové řešení, tak samotné řešení tohoto projektu procházelo třemi fázemi:

1. prototyp,
2. produkční služba generující lístky a zastřešující komunikaci s externími *API*,
3. prezentační web.

5.1 Prototyp

První fáze zadání se zabývala prototypem služby. Cílem tohoto prototypu bylo vytvořit jednoduchou implementaci, jenž měla za cíl pro jednu z platforem a na základě poskytnutých dat, vygenerovat lístek a následně jej vzdáleně aktualizovat.

V rámci prototypu jsem měl možnost zmapovat technologie a prostředky, potřebné pro implementaci komplexnějšího řešení. V této fázi nebyly kladeny žádné požadavky na architekturu, spíše v relativně rychlém čase odpovědět na otázku, zda a jak je tento problém možný řešit.

Jedním z výstupů tohoto prototypového projektu bylo i rozhodnutí, zda v další fázi implementovat *in-house* řešení generování lístků, nebo integrovat jednu ze služeb třetích stran. V zadání prototypu byl explicitně specifikován požadavek vyzkoušet generování lístků jako *in-house* službu na základě jejíž složitosti se provede zmíněné rozhodnutí.

5.1.1 Analýza

Pro prototyp byla vybrána platforma aplikace *Apple Wallet* od společnosti Apple, tudíž předmětem analýzy bylo zejména nastudování dokumentace k *Apple Wallet* lístkům [4.2] a na základě těchto poznatků navrhnout řešení.

V rámci analýzy jsem ještě provedl prozkoumání možností externích knihoven pro generování lístků a z dat o aktuálnosti dostupných knihoven, z jednotlivých GitHub repositářů, jsem vybral pro řešení knihovnu *dotnet-passbook* [11].

5.1.2 Technická specifikace

Prototyp jsem implementoval jako *REST API* řešení na platformě .NET Core 3.1 v jazyce C# 8.0 s využitím NuGet balíčku *dotnet-passbook*, a aplikace Postman pro testování *API*.

dotnet-passbook

Jedná se o NuGet balíček, který poskytuje služby a objekty nejen pro generování lístků pro platformu *Apple Wallet*, ale také pro jejich podepsání příslušnými certifikáty a vygenerování distribuovatelného souboru [4.2.4].

Tato knihovna je postavena na platformě .NET Standard 2.0, tudíž ji lze referencovat v projektech na všech platformách .NET.

Autor této knihovny také vytvořil webovou aplikaci *pkpassvalidator*, která má za úkol validovat vygenerované *pkpass* soubory. Lístek označený tímto nástrojem jako validní, je považován za zobrazitelný na Apple mobilním zařízení.

5.1.3 Implementace

Z architektonického hlediska byla tato fáze pojata nejjednodušším možným způsobem, kdy celá implementace byla součástí jednoho .NET Core 3.1. WEB API projektu. Implementace byla rozdělena na 3 separátní části, z nichž každá se skládala z *API* řadiče, jenž obsluhoval *HTTP* požadavky na ním implementované koncové body a objektu služby, jenž vykonávala logiku nutnou k úspěšnému vykonání požadované operace.

Generování lístků

Požadavky na generování lístků obstarával *API* řadič `PassBookController`. Logiku generování lístku obstarávala C# třída `PassGeneratorService`, která sama využívala objekty a služby z knihovny *dotnet-passbook*.

Třída `PassGeneratorService` měla za úkol vytvořit a naplnit daty datový objekt lístku `PassGeneratorRequest` z knihovny *dotnet-passbook*, který nakonfigurovala dle šablony *event ticket*

[4.2]. Tento objekt byl následně vložen do metody, `Generate` třídy `PassGenerator` z knihovny *dotnet-passbook*, který z něj vygeneroval výsledný podepsaný lístek, jako pole bajtů.

Toto pole bajtů sloužilo jako výsledek z metody `GetPass` třídy `PassGeneratorService` a následně bylo vráceno, jako *MIME* typ *application/vnd.apple.pkpass*, z koncového bodu *API* `GetPass`, běžícího na podadrese */pass*.

Komunikace zařízení se serverem

Komunikaci zařízení se serverem měl na starosti *API* řadič `PassKitController`. Název řadiče byl zvolen záměrně podle frameworku *PassKit* od společnosti Apple. V tomto řadiči jsem implementoval rozhraní popsané v kapitole 4.2.3 na straně 18.

V rámci komunikace je nutné si držet informace o zařízení, registracích a jednotlivých lístcích a protože jsem ve fázi prototypu pracoval jen s jednou instancí lístku, tak jsem persistenci těchto dat řešil pouze v paměti aplikace.

Aktualizace lístku

Pro aktualizaci lístku jsem implementoval koncový bod `UpdatePass`, běžící na podadrese služby */pass/notification*. Tento koncový bod nepřijímal žádné parametry a za úkol měl pro lístek v paměti, zaslat *push* notifikaci na zařízení, jenž byla k lístku zaregistrována.

Zaslání notifikace jsem řešil pomocí tříd `TcpClient` a `SslStream`. Skrze instanci třídy `TcpClient` jsem vytvořil spojení se serverem obsluhujícím Apple *push* notifikace, běžícím na adrese serveru *gateway.push.apple.com* a síťovém portu *2195* a dále jej autentizoval pomocí třídy `SslStream` a certifikátu, přiřazenému distributorovi lístků. Skrze toto navázané spojení jsem zaslal *push* notifikaci pro každý přiřazený `pushToken` zařízení.

Při testování procesu aktualizace lístku jsem se setkal s problémem, kdy zařízení nekontaktovalo lokální server z důvodu nedůvěryhodnosti. Při zkoumání tohoto problému jsem zjistil, že tuto komunikaci lze efektivně ladit pouze za použití počítače od společnosti Apple a vývojového prostředí *xCode*, které je schopno simulovat důvěryhodný server. Dalším řešením je aplikaci provozovat na veřejném serveru s nainstalovaným Apple *WWDR* certifikátem. Pro účely testování této komunikace a prezentace prototypu jsem využil firemní notebook od společnosti Apple.

5.2 Produkční implementace

V prototypové fázi jsme si ověřili, že řešení pomocí vlastní implementace je vyhovující a samotný projekt byl schválen pro produkční implementaci. Na základě úspěšného prototypu jsem v této fázi implementoval již službu pro produkční využití. Výsledkem této implementace mělo být kompletní řešení generování a distribuce lístků pro *Apple Wallet* a *Google Pay*. Dále pak jejich aktualizace a v případě modulu pro *Apple Wallet*, tak také jejich persistence.

Po úspěšném nasazení služby do produkce, bylo dále součástí zadání integrace služby do administrátorské aplikace, kvůli správy vygenerovaných lístků.

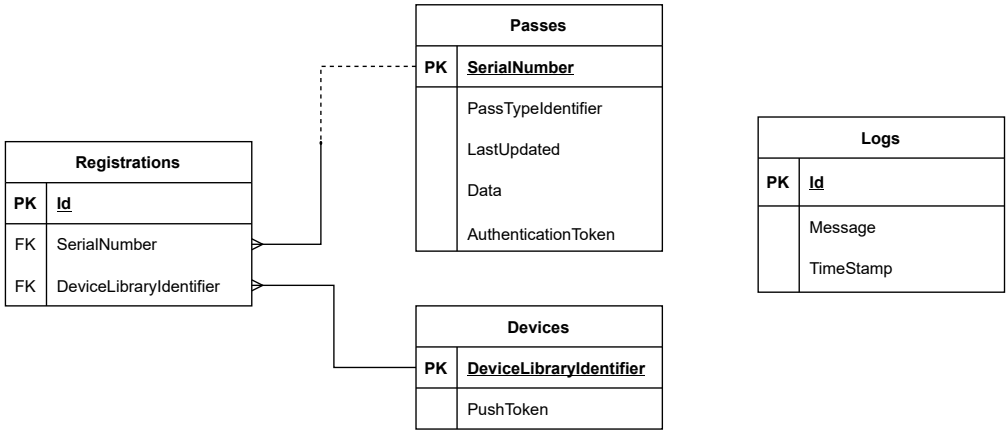
5.2.1 Analýza

Produkční implementace již vyžadovala rozsáhlejší analýzu jak integrační části, tak zejména z hlediska architektury a celkového návrhu systému [5.3]. Součástí tohoto návrhu byla také procesní část vývoje software a to časový odhad implementovaného řešení.

Integrace

Jelikož se při analýze a následné implementaci prototypu osvědčilo řešení pro *Apple Wallet* pomocí knihovny *dotnet-passbook*, tak jsem pro tento modul zvolil podobný přístup i pro produkční implementaci.

Pro integraci služby *Apple Wallet* je nutné si držet data o lístcích, zařízeních a jejich registracích na straně distributora a použité úložiště v paměti aplikace z prototypu již nebylo dostatečné. Jelikož se jednalo o pouhé čtyři různé entity, z nichž tři vyžadovaly jistou míru provázanosti, tak jsem navrhl jednoduchý relační model pro následnou implementaci pomocí relační databáze.



Obrázek 5.1: Relační model databáze pro uchování dat k lístkům pro *Apple Wallet*

Největší část analýzy byla tvořena analýzou integrace lístků pro *Google Pay*, kterou popisuji v kapitole 4.3 na straně 22. Jelikož se jedná o integraci cloudové služby, která veškeré data o vygenerovaných lístcích držela právě v cloudovém uložišti služby *Google Pay*, tak po shodě s vedením projektu bylo rozhodnuto, že není tedy nutné uvažovat řešení úložiště na straně serveru a že není nutné ukládat reference na vygenerované Google lístky na straně distributora.

Po prozkoumání dokumentace k službě *Google Pay API for Passes* [4.3.1] jsem se rozhodl pro integraci využít knihovny přímo poskytované společností Google. Pro autentizaci aplikace vůči Google serverům jsem využil knihovnu *Google.Apis* a zejména její část *Google.Apis.Auth*, které jsou

dostupné skrze balíčkovací systém pro platformu .NET, *NuGet Package Manager*. Pro samotné generování lístků jsem pak využil klientskou knihovnu *Google.Apis.Walletobjects.v1*, která je dostupná pouze jako stažitelný adresář s rozhraním pro komunikaci a objekty pro tvorbu lístků.

Architektura

Z pohledu architektury bylo požadavkem implementovat do řešení prvky návrhového vzoru *DDD - Domain Driven Design*, který je již úspěšně využit na ostatních službách v rámci projektu London Theatre Direct a prvky návrhového vzoru *CQRS - Command and Query Responsibility Segregation.*

Domain Driven Design

Domain Driven Design [12], nebo-li také *DDD* je návrhový vzor vývoje software, založený na jasně specifikovaných a oddělených doménách *business* logiky daného systému. Z hlediska procesu vývoje software se jedná spíše o analytický pohled na věc, díky kterému jsme schopni tyto domény správně navrhnout a následně implementovat.

Pojmem doména můžeme rozumět takovou část logiky, která jednoznačně specifikuje účel implementovaného software. Tuto doménu v kódu reprezentujeme jejím modelem, který by měl být co nejpřesnějším odrazem reality. V ideálním případě by tento model měl obsahovat, kromě dat, také logiku, která zajišťuje komplexní funkčnost domény jako celku. Při modelování domény je zejména nutné dbát na platformní a datovou nezávislost. Zjednodušeně řečeno, správně navržená doména by měla být znovupoužitelná a nezávislá na implementačních detailech, či datovém zdroji, který ji poskytuje data.

Doménový návrh je úzce spjat s objektově orientovaným programováním. Každá komplexnější doména je pak rozdělena do doménových objektů.

- ***Aggregate Root***, jak je již patrné z anglického slovíčka *root*, jedná se o tzv. kořen domény. Jako *aggregate root* nazýváme nejen kořenový objekt domény, ale také samotnou oddělenou část domény (např. v adresářové struktuře doménového projektu). Tento objekt je identifikován systémovým identifikátorem (např. identifikátor z databáze) a dále na sebe váže další objekty z dané domény. Především však obsahuje doménovou logiku. V našem případě by tímto kořenem domény mohl být lístek do divadla.
- ***Value Object***, tedy hodnotový objekt je jasně identifikován daty jenž v sobě drží a zpravidla neobsahuje logiku, nebo pouze validační. Typickým příkladem hodnotového objektu by mohl být objekt držící informace o adrese divadla.

Aggregate root v kontextu oddělené části domény neobsahuje ovšem jen objekty, ale také rozhraní služeb, skrze která komunikuje s ostatními částmi aplikace. Tyto rozhraní služeb se dělí na tři typy, doménové služby, aplikační služby a infrastrukturní služby. Doménové rozhraní implementujeme v samotné doménové vrstvě aplikace a takto implementovaná služba slouží zejména pro logiku,

která si vynucuje práci nad vícero doménami. Zpravidla se těmto službám chceme pokud možno vyhýbat, jelikož každý *aggregate root* by měl být nezávislý na jiných doménách. Aplikační služby implementujeme v aplikační vrstvě aplikace. Tyto služby obsahují, jak již jejich název napovídá, aplikační logiku, které řeší většinou platformně závislé úkony, či využití knihoven specifických pro danou aplikaci. Jako poslední implementujeme infrastrukturní služby, které implementujeme v infrastrukturní vrstvě a které slouží zejména pro komunikaci s externími službami, či datovými zdroji aplikace.

Z hlediska návrhu domény rozlišujeme dva typy doménových modelů. *Domain Rich Model*, doménově bohatý model a *Anemic Model*, tzv. anemický, čili chudý model. Bohatým modelem označujeme doménové návrhy obsahující drtivou většinu logiky právě v doméně, naopak *Anemic Model* má většinu logiky řešenou skrze aplikační, či infrastrukturní služby a doménové objekty slouží převážně pro držení dat.

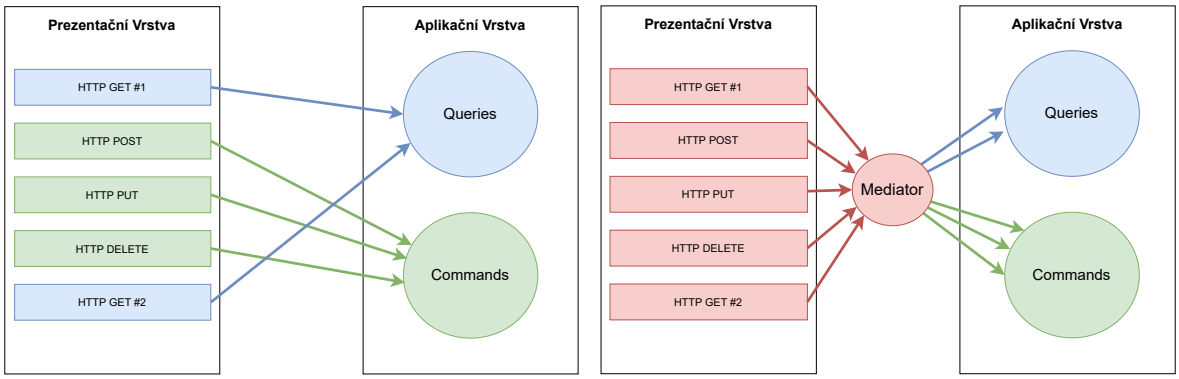
CQRS - Command and Query Responsibility Segregation

CQRS [13] je návrhový vzor, jenž má za úkol oddělit čtecí a zapisovací části aplikace. Tento návrhový vzor jde ruku v ruce s návrhem webových *API*. S těmito službami komunikujeme na principu požadavků, které, pokud jde např. o *REST API*, lze rozdělit na *HTTP Get, Post, Put, Patch, Delete* požadavky. Tuto skupinu požadavků lze dále rozdělit na dvě separátní části, čtecí a zapisovací (manipulační).

Čtecí část tvoří *HTTP GET* požadavek. Tato část se vyznačuje požadavkem na rychlost zpracování. Většinou se jedná jen o jednoduché přečtení dat z datového zdroje, pokud zahrnuje i zápis, zejména požadavky na data, která ještě neexistují, tak je řešen pomocí tzv. *quick path*. Pokud data existují, tak jsou navrácena přednostně, až pokud je zjištěno, že data neexistují, tak je posléze řešen zápis. Čtecí požadavky můžeme souhrnně označit anglickým pojmem *queries*, tedy v překladu, dotazy. Tyto dotazy fungují zpravidla na principu požadavek-odpověď, tedy na každý dotaz by měla přijít odpověď.

Manipulační část tvoří zbývající *HTTP* požadavky. Tyto požadavky řeší výhradně zápis, nebo manipulaci již existujících dat. Tato manipulace vyžaduje zpravidla složitější infrastrukturu pro provedení manipulace, která není třeba pro jednodušší dotazy, z tohoto důvodu tyto části oddělujeme. Manipulační požadavky označujeme pojmem *commands*, příkazy.

Implementace *CQRS* přístupu úzce souvisí s návrhovým vzorem *Mediator*. Tento návrhový vzor přidává do výsledné implementace tzv. prostředníka. Skrze tohoto prostředníka probíhá komunikace mezi prezentační vrstvou a zbytkem aplikace, kdy prezentační vrstva nemá přímou závislost na implementaci obsluhující daný požadavek, ale je závislá pouze na prostředníkovi, kterému předá požadavek a prostředník jej přesměruje na místo určení.



(a) Bez prostředníka (b) S prostředníkem

Obrázek 5.2: Implementace *CQRS* návrhového vzoru

5.2.2 Technická specifikace

Produkční službu jsem implementoval ve verzi jazyka C# 9.0 a platformě .NET 5. Ačkoliv .NET 5 není verze s dlouhodobou podporou, tak jsme se ji rozhodli využít, zejména kvůli vyzkoušení nových možností platformy .NET a následné jednodušší migraci na příští *LTS* verzi, .NET 6.

Pro persistentní uložiště jsem využil Microsoft SQL Server a pro jeho propojení s aplikací objektově-relační mapovací framework Entity Framework Core ve verzi 5.0.2. Pro aplikační implementaci jsem využil knihoven *MediatR*, *AutoMapper*, *dotnet-passbook* [5.1.2] a zmíněných *Google.Apis*.

MediatR

Knihovna *MediatR* [14] poskytuje implementaci návrhového vzoru *mediator*. Pomocí této knihovny je řešena komunikace mezi prezentační vrstvou a zbytkem aplikace. Knihovna poskytuje řadu rozhraní pro implementaci z nichž stěžejní jsou rozhraní `IRequest` a `IRequestHandler`.

Rozhraní `IRequest` implementuje objekt požadavku. Rozhraní je generické a jako generický parametr mu předáváme typ odpovědi na požadavek. Každá implementace rozhraní `IRequest` má odpovídající implementaci generického rozhraní `IRequestHandler`, kterému jako generické parametry předáváme přidruženou implementaci požadavku a typ odpovědi. Zatímco implementace požadavku obsahuje datové vlastnosti specifické pro daný požadavek, tak tzv. *handler*, objekt obsluhující požadavek, obsahuje logiku pro úspěšné vyřešení požadavku.

Tato knihovna hojně využívá princip *IoC - Inversion of Control*, kdy jednotlivé objekty jsou při startu aplikace zaregistrovány do tzv. *DI* kontejneru. Který pak poskytuje instance objektů na vyžádání skrze konstruktor, metodu či vlastnost objektu. Knihovna *MediatR*, využívá tohoto principu právě díky implementacím poskytnutých rozhraní.

Celý proces využití prostředníka skrze tuto knihovnu začíná zavoláním metody `Send` na instanci objektu `Mediator`. Parametrem této metody je objekt implementující rozhraní `IRequest` a na základě poskytnutého objektu, jenž je také generickým parametrem rozhraní `IRequestHandler`, je schopen rozhodnout, kterou implementaci rozhraní `IRequestHandler` má s požadavkem oslovit.

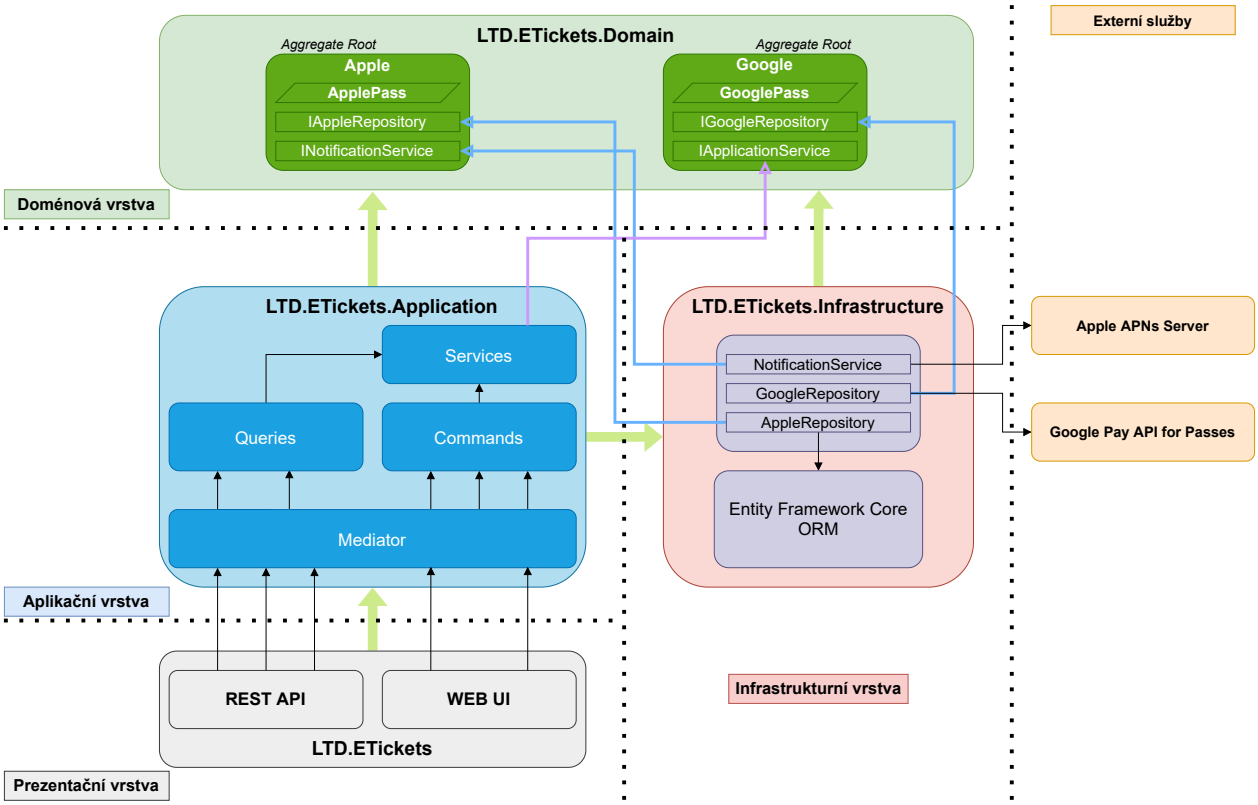
AutoMapper

AutoMapper [15] je knihovna ulehčující přelévání dat z jednoho objektu do objektu druhého. Mapování probíhá na základě shody jmen vlastností objektů, tudíž např. vlastnost `FirstName` ze zdrojového objektu je automaticky mapovatelná na vlastnost `FirstName` z objektu cílového.

Skrze tzv. profily lze této knihovně specifikovat složitější mapování mezi objekty, díky kterým je následně umožněno upustit od specifických mapovacích metod v kódu a mít separátně specifikované kontrakty mezi objekty různých vrstev aplikace.

5.2.3 Implementace

Implementaci služby jsem začal vytvořením projektové struktury dle návrhu [5.3].



Obrázek 5.3: Návrh služby London Theatre Direct ETickets

Prezentační vrstvu *LTD.ETickets* jsem vytvořil jako .NET 5 Web *API* projekt. Aplikační, doménovou a infrastrukturní vrstvu jsem vytvořil jako knihovny tříd .NET 5. Závislosti mezi jednotlivými projekty jsou znázorněný zelenými šipkami, také na obrázku 5.3. Dále se implementace dělila do fází dle jednotlivých integrací a implementace jednotlivých vrstev aplikace.

5.2.3.1 Apple

V první fázi bylo nutné navrhnout doménu aplikace. Jak jsem se již zmiňoval v rámci povídání o *DDD* na straně 30, doménu aplikace tvoří logika nebo její předpis, již má daná aplikace řešit. V našem případě se jedná o distribuci lístků.

V rámci analýzy a zkušeností z prototypu jsem se snažil nalézt společné prvky mezi integrací pro *Apple Wallet* a *Google Pay*, které by umožnili navrhnout doménu společnou pro obě implementace. Bohužel jediné společné prvky se objevily pouze pokud se jednalo o proces aktualizace lístku, tudíž jsem navrhl dva separátní doménové kořeny, pro každou integraci zvlášť.

Doménová vrstva

Středobodem domény pro integraci služby *Apple Wallet* byl objekt `ApplePass`. Tento objekt je předmětem veškerého datového toku v rámci integrace *Apple Wallet*. Obsahuje identifikátory samotného lístku, datum poslední změny, data ve formátu *JSON*, vygenerovaný soubor ve formě bajtového pole a *push* token každého zařízení, jenž je k lístku zaregistrováno. Další objekty, které jsem vytvořil v doméně jsou již jen hodnotové objekty `PassRegistrationResult` a `SerialNumbersResponse`, které sloužili pouze pro jednodušší čtení dat z databáze v rámci komunikace zařízení se serverem [4.2.3].

Tyto objekty jsou vytvářeny v infrastrukturní vrstvě aplikace, a to skrze implementaci rozhraní repositáře `IAppleRepository` z vrstvy doménové. Toto rozhraní poskytuje metody pro načtení a zápis dat do databáze, a jelikož se tato implementace z velké části shoduje s prototypem, tak jsem již v této fázi věděl, které metody v rozhraní deklarovat. Podobně jsem navrhl rozhraní `IUpdateNotificationSender`, jehož implementace měla za úkol odeslat *push* notifikace na zařízení uživatele.

Posledními částmi domény pro tuto integraci bylo validační rozhraní `IValidator` s příslušným hodnotovým objektem, jenž nesl výsledek validace lístku a rozhraní `IApplePassGenerator`, jehož implementace měla za úkol vytvoření lístku, jeho podepsání příslušnými certifikáty a vygenerování souboru, jenž mohl být dále distribuován. Ač se zdá, že logika zařizující tyto procesy by měla spadat přímo do domény, není tomu tak. Z důvodu úzké závislosti na knihovně *dotnet-passbook*, je tato logika vykonávána na aplikační vrstvě jako součást aplikačních služeb.

Infrastrukturní vrstva

Další fází implementace byla infrastrukturní vrstva. V této fázi jsem za prvé musel vytvořit databázi a její model v aplikaci. Jelikož se jedná o novou databázi a pro implementaci jsem využil

Entity Framework Core, tak jsem využil tzv. *code-first* přístup, kdy prvně vytvoříme databázové entity a databázový model jako C# objekty a pak skrze databázové migrace vygenerujeme fyzický databázový model. Relační model lze vidět na obrázku 5.1 na straně 29.

Vytvořený *code-first* model rozšiřuje třídu `DbContext` z frameworku *Entity Framework Core*. Skrze tento databázový kontext, který je připojen k databází přes tzv. `connectionString`, jehož hodnotou jsou připojovací údaje (server, přihlašovací údaje atd.), jsme schopni načítat a zapisovat data do dané databáze. Databázový kontext je hlavní závislostí implementace rozhraní `IAppleRepository`, jenž má za úkol právě tyto databázové operace. Tyto operace jsem implementoval za pomocí nadstavby jazyka *LINQ*, *LINQ to Entities*, která nám umožňuje psát dotazy do databáze skrz *Entity Framework Core* zápisem v jazyce *LINQ*.

```
async Task<ApplePass> IAppleRepository.GetPass(string serialNumber)
{
      var pass = await dbContext.Passes.Where(x => x.SerialNumber == serialNumber)
                              .Select(x => new ApplePass
                              {
                                    AuthenticationToken = x.AuthenticationToken,
                                    Data = x.Data,
                                    LastUpdated = x.LastUpdated,
                                    PassTypeIdentifier = x.PassTypeIdentifier,
                                    SerialNumber = x.SerialNumber,
                                    PushTokens = x.Registrations
                                            .Select(x => x.Device.PushToken)
                                            .ToList()
                              }).FirstOrDefaultAsync();
      return pass ?? ApplePass.Empty;
}
```

Výpis kódu 5.1: Příklad zápisu infrastrukturní metody za využití jazyka *LINQ*

Jako poslední jsem v rámci infrastruktury pro *Apple Wallet* implementoval rozhraní `IUpdateNotificationSender`, jehož implementace byla téměř totožná s implementací stejné logiky v prototypu aplikace [5.1].

Je nutné podotknout, že veškeré implementace infrastrukturních a aplikačních služeb jsou opatřeny modifikátorem přístupu `internal` a všechny metody jsou explicitně implementovány přes dané rozhraní. Tímto je zajištěn přístup k implementaci těchto služeb pouze přes rozhraní deklarována v doméně aplikace s modifikátorem `public`.

Aplikační a prezentační vrstva

V aplikační vrstvě bylo jako první nutné nainstalovat potřebné *NuGet* balíčky, *MediatR*, *AutoMapper* a *dotnet-passbook*. Aplikační vrstvu jsem dále přizpůsobil dle požadavků kladených při implementaci *CQRS* návrhového vzoru. Vytvořil jsem tedy adresářovou strukturu rozdělující *queries*, dotazy a *commands*, příkazy. Dále jsem vytvořil složku *Services*, ve které jsem implementoval rozhraní z doménové vrstvy `IApplePassGenerator` a `IValidator` a vytvořil jsem složku *Models* ve které jsem uložil modely (kontrakty) vystupující v rámci komunikace zařízení se serverem [4.2.3].

Jelikož implementace aplikační vrstvy je, mimo aplikační služby, úzce spjata s implementací prezentační vrstvy, tak jejich implementace probíhala paralelně. Pro doménu *Apple* jsem vytvořil *API* řadič, který implementoval mimo samotný požadavek na generování lístků, také rozhraní předepsané pro komunikaci se zařízením [4.2.3].

Pro každý požadavek na řadič jsem vytvořil *query*, nebo *command*, objekt do příslušného adresáře v aplikační vrstvě a v rámci stejného C# souboru jsem vytvořil i příslušný *handler* objekt. Skrze tento *handler* a jeho závislost na repositáři či jiných službách deklarovaných jako rozhraní v doménové vrstvě je prováděna příslušná logika.

```
public class UpdatePassCommand : IRequest<PassChangedResponse>
{
    //Vlastnosti příkazu UpdatePassCommand
}
public class UpdatePassCommandHandler
   : IRequestHandler<UpdatePassCommand, PassChangedResponse>
{
      public async Task<PassChangedResponse> Handle(UpdatePassCommand request)
      {
         //Logika obsluhující příslušný command
      }
}
//Volání příslušného command objektu z řadiče skrze knihovnu MediatR
PassChangedResponse response = await Mediator.Send(new UpdatePassCommand(...));
```

Výpis kódu 5.2: Příklad implementace *CQRS* za využití knihovny *MediatR*

Pro koncové body obsluhující dotazy, či příkazy vyžadující autorizaci jsem implementoval speciální rozhraní `IPipelineBehavior`, které přijímá stejné generické parametry jako rozhraní `IRequestHandler` (viz. povídání o knihovně *MediatR* na straně 32). Implementace tohoto rozhraní je zavolána vždy před nebo po vykonání logiky implementované v *handler* objektu. Těchto speciálních chování můžeme pro každý dotaz, či příkaz, implementovat nespočet a pořadí jejich vykonávání je závislé na pořadí registrace do *DI* kontejneru. Jelikož je implementace tohoto autorizačního chování

společná pro několik požadavků, příslušné *query* a *command* objekty jsem seskupil pod rozhraní `IAppleRequest`, díky kterému jsem mohl naimplementovat generickou implementaci autorizačního rozhraní.

Nejdůležitějším koncovým bodem, implementovaným v rámci řadiče `AppleController`, je `GetAppleETicket` koncový bod. Jako parametry přijímá identifikátor košíku a lístku z prodejního systému a na základě těchto validně zadaných identifikátorů je návratovou hodnotou samotný soubor vygenerovaného lístku [A.1a], jenž je přidán do aplikace *Apple Wallet*.

5.2.3.2 Google

Jelikož se u řešení pro *Google* jedná zejména o bránu k *Google API for Passes* je zde doménová vrstva výrazně tenčí, než v případě implementace pro Apple a většina logiky je zde implementována v infrastrukturní vrstvě.

Doménová vrstva

Středobodem domény je doménový objekt lístku, v tomto případě `GooglePass`. Tento objekt v sobě nese hodnotový objekt `GooglePassIdentifier`, který specifikuje identifikátory daného lístku. Dále nese informaci o validitě lístku a pokud je označen za validní, tak vygenerovaný `JwtToken`, nutný pro následnou distribuci. Dále pak sám obsahuje vlastnost generující distribuční odkaz na lístek.

V rámci domény jsou ještě dostupné konfigurační objekty, které v sobě nesou data o distributorovi a jejich dostupnost je nutná v celé aplikaci.

Jedinou službou, které rozhraní doména specifikuje, je služba `GooglePassService`, v jejímž rozhraní `IGooglePassService` jsem nadeklaroval metody pro načtení, vytvoření, zrušení a aktualizaci lístku.

Infrastrukturní vrstva

Infrastrukturu pro integraci *Google* řešení jsem započal integrací `Google.Apis.Walletobjects` klienta poskytnutého společností *Google*. Tento klient není distribuován skrze *NuGet* balíčkovací systém, ale pouze jako stažitelný kód, tudíž jsem jej přidal do projektu jako další projektový soubor.

Nad tímto klientem jsem vytvořil abstrakci `GoogleClient` s rozhraním `IGoogleClient`. Tato abstrakce využívala metody a poskytovala objekty z klienta `Google.Apis.Walletobjects`.

Samotným jádrem této infrastruktury byla implementace doménového rozhraní infrastrukturní služby `IGooglePassService`. Tato služba zajišťovala generování a obecnou správu lístků a komunikovala skrze mnou implementovaným klientem `GoogleClient` s externími servery. Pro práci s daty v metodách této služby jsem vytvořil tzv. kontraktovou třídu. Ta měla za úkol držení dat lístku v podobě objektů `EventTicketClass` a `EventTicketObject` [4.3.2]. Tyto objekty jsem vytvářel pomocí tzv. rozšiřujících metod (*extension* metody), které pracují s danou instancí třídy a pro data předané parametrem tuto třídu naplní.

```
private static void Build(this EventTicketClass ticketClass, PassPayload payload)
{
   ticketClass.Identifier = payload.PerformanceId;
   ticketClass.Message = new Message
   {
      Title = payload.EventName,
      Content = payload.Description,
      Timeout = TimeSpan.FromSeconds(15)
   };
}

eventTicketClass.Build(passPayload);
```

Výpis kódu 5.3: Příklad implementace *extension* metody pro instanci objektu `EventTicketClass` a parametrem typu `PassPayload`

Aplikační a prezentační vrstva

Prezentační vrstvu tvořil opět jeden specifický řadič dle domény, `GoogleController`. Jelikož zde je komunikace mezi zařízením a lístkem pouze na bázi vyžádání lístku, tak se v celém řadiči nachází jen jeden koncový bod na tento požadavek. Tento koncový bod podobně jako u vyžádání lístku pro *Apple Wallet* přijímá jako parametry identifikátor košíku a identifikátor lístku z prodejního systému. Odpovědí na tento požadavek byla adresa ke stažení vygenerovaného lístku [A.1b].

V aplikační vrstvě jsem vytvořil patřičný *query* objekt, který byl skrze prostředníka obsluhován příslušným *handler* objektem. Jelikož objekt požadavku musí být na straně serveru obohacen o identifikátory distributora (pro proces generování lístku), tak jsem i zde implementoval rozhraní `IPipelineBehavior`, které před inicializací *handler* objektu, tyto data přidalo k požadavku. Jelikož jsem v této fázi věděl, že při implementaci řadiče a požadavku na aktualizaci lístků bude potřeba obohacovat tyto požadavky stejným způsobem, tak jsem implementaci tohoto chování znovu vytvořil jako generickou a požadavky pro integraci Google jsem seskupil pod rozhraním `IGoogleRequest`.

5.2.3.3 Obecná implementace

Neméně důležitou částí jako samotné integrační části projektu byla i obecná implementace. V této fázi jsem implementoval zejména doménové rozhraní `ILtdRepository`.

Tato infrastrukturní služba pokrývala veškeré čtení a validaci dat o zakoupených lístcích přes portál *londontheatredirect.com*. Tyto data jsem seskupil pod objektem `PassPayload`, který dodával data, jak pro Apple, tak pro Google integraci. Tento repositář měl přímou závislost na databázovém kontextu s modelem databáze pro celý prodejní systém.

V této službě jsem také implementoval validační metody pro validaci požadavků na vygenerování lístků. Tyto validace se prováděly společnou implementací generického rozhraní `IPipelineBehavior` v aplikační vrstvě, které validuje, zda požadovaný lístek vůbec existuje a zda je označen jako generovatelný.

Aktualizace lístku

V rámci této logiky jsem pracoval s instancemi implementací rozhraní z doménové vrstvy aplikace `ILtdRepository`, `IAppleRepository`, `IGooglePassService`, `IApplePassGenerator` a `IUpdateNotificationSender` poskytnutými z *DI* kontejneru. Pro každou z integrací jsem si pro identifikátor lístku z prodejního systému načetl vygenerované lístky pro Apple a Google. Na základě dat z repositáře `LtdRepository` jsem příslušné lístky upravil a následně jejich upravenou verzi, pro Google, odeslal skrze rozhraní `IGooglePassService` na *Google Pay API for Passes* a pro Apple, uložil upravenou verzi v relační databázi a odeslal *push* notifikaci zákazníkovi.

Pro aktualizaci lístku jsem implementoval *API* řadič `PassController`, který implementoval koncový bod `UpdatePass`. Tento koncový bod konzumující *HTTP Post* požadavek na podadrese */api/pass/notification* přijímal v těle požadavku interní identifikátor lístku, interní identifikátor košíku a výčtový typ `DigitalPassStatus`, který obsahoval stavy `Updated` a `Cancelled` pro příslušné operace nad lístkem. Objekt tvořící tělo tohoto požadavku, byl vystaven jako kontraktový objekt `PassChanged`. Pro tento koncový bod byl vystaven příslušný *command* objekt `UpdatePassCommand` a jeho obsluhující *handler* objekt v němž se veškerá aktualizační logika prováděla.

Integrace do administrace prodejního systému

Tato implementace nebyla kompletně v mé kompetenci a při plnění toho úkolu jsem spolupracoval s kolegy pracujícími na administračním projektu.

V rámci této integrace bylo nutné při jakékoliv úpravě lístku kontaktovat implementované *REST API* na podadrese */api/pass/notification*, které daný lístek aktualizovalo. V rámci této komunikace bylo nutné zaslat kontraktový objekt `PassChanged` dle předpisu popsaném v předchozí podkapitole. Předpis pro tento objekt a url adresu koncového bodu jsem kolegům předal k integraci na požadovaných místech v rámci administrační aplikace.

5.3 Webová prezentace

Součástí zadání a zároveň poslední fází projektu byla prezentační webová stránka, na které si zákazník mohl vyžádat vygenerování lístku. Tato webová prezentace byla zakomponována do zmíněné přestavby portálu *londontheatredirect.com*, kdy bylo úkolem připravit použitelnou šablonu [B.1] pro prezentaci lístků v duchu nového návrhu prodejního portálu.

5.3.1 Technická specifikace

Backend, tedy infrastrukturu webu, jsem implementoval v technologii *Razor Pages*, která umožňuje kombinovat C# kód s *html* kódem v rámci speciálních *cshtml* souborů a knihovny *React.AspNet*, která poskytovala propojení *backendu* jednotlivých stránek a jejich prezentační části. *Frontend*, prezentační část, jsem implementoval v HTML, za použití CSS stylů a následně za využití *frontend* frameworku *ReactJS* [16].

Celá prezentační část byla implementována v typové nadstavbě programovacího jazyka *JavaScript*, jazyce *TypeScript* [17]. Propojení mezi prezentační a infrastrukturní částí zajišťovala knihovna *TypeGen*, díky které lze generovat kontrakty mezi prezentační a infrastrukturní částí z C# tříd do tříd, či rozhraní v jazyce *Typescript*. Jako zdroj knihoven pro *React* implementaci jsem zvolil balíčkovací systém *Yarn*, který mi poskytl knihovnu *styled-components* [18] pro stylování *React* komponent v rámci *TypeScript* kódu.

5.3.2 *Backend*

Součástí infrastruktury webové prezentace byla v první fázi specifikace modelů stránek. Jelikož požadavky na generování lístků přichází pro celou zaplacenou objednávku, tzv. košík, bylo nutné specifikovat nejen model pro lístek, ale také pro představení, ke kterému lístek náleží. Specifikace modelů pro stránku vycházela z návrhu prezentační stránky (příloha B.1). Z těchto modelů jsem za pomoci *NuGet* balíčku *TypeGen* vygeneroval rozhraní v jazyce *Typescript*. Tato rozhraní plnila úlohu tzv. kontraktu a udržovala konzistenci v přenosu dat z infrastruktury na prezentační část webu.

Načtení dat pro prezentaci jsem implementoval, stejně jako u služby pro generování lístků, pomocí knihovny *MediatR* a již implementované infrastrukturní služby `LtdRepository`.

Generování stránky i s daty jsem implementoval na straně serveru, tudíž do konzumujícího prohlížeče již putovala již naplněná stránka. Propojení mezi *ASP.NET backend* implementací a *ReactJS* prezentací jsem zajistil, díky knihovně *React.AspNet*. Tato knihovna na základě tzv. stránkovacích konvencí, kdy na základě cesty k *React* souboru v rámci projektu, jsou přenesena data z *Razor Pages backendu* na *frontend*. Pro tento datový tok je stěžejní konzistence již zmíněných kontraktů.

5.3.3 *Frontend*

První verzi webové prezentace jsem implementoval jako *html* stránku za využití syntaxe technologie *ASP.NET Razor Pages*. V další fázi jsem již využil *frontend* framework *ReactJS*, který usnadnil zejména znovupoužitelnost jednotlivých částí stránky.

Jelikož framework *ReactJS* je založen na tzv. komponentách, kdy se návrh stránky rozkládá na znovupoužitelné části, které se implementují jako samostatné celky a výsledný produkt se z nich jen skládá, musel jsem návrh webové prezentace [B.1] analyzovat a do těchto komponent rozdělit.

Komponenty jsem rozdělil do dvou skupin:

1. **Kontejnery** - bloky obalující sadu atomických komponent,
2. **Komponenty** - atomické komponenty, zobrazující danou entitu (např. lístek).

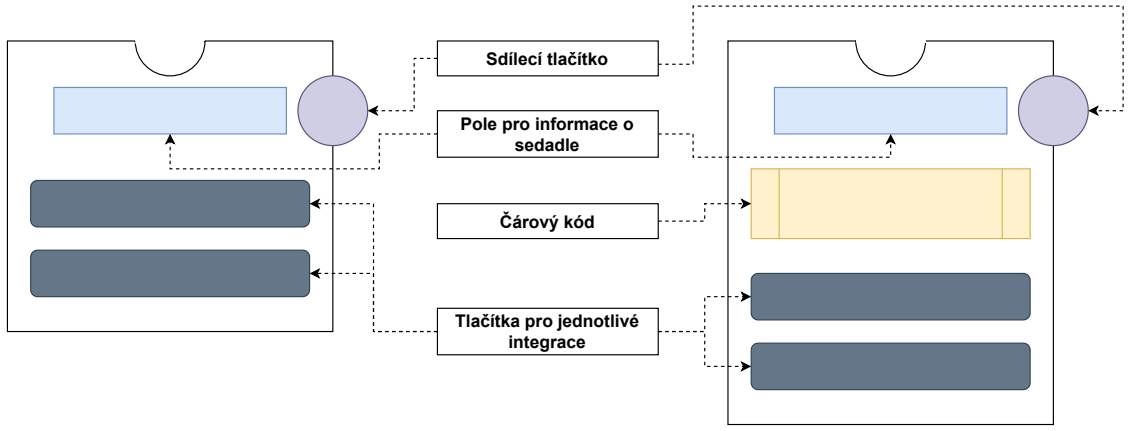
Samotné komponenty jsou psány v nadstavbě jazyka *JavaScript* v tzv. *JSX* (při využití jazyka *TypeScript*, *TSX*), která umožňuje kombinovat tradiční zápis *JavaScript* kódu s *html* kódem. Takhle složený kód je při běhu aplikace překládán do *JavaScript* kódu.

Každou komponentu jsem následně implementoval jako funkcionální. Framework *ReactJS* umožňuje implementoval komponenty dvěma způsoby. Prvním, dnes už neprosazovaným, způsobem je implementace pomocí třídy, která rozšiřuje bázovou třídu `React.Component`. Tato třída musí implementovat metodu `render`, jejíž návratovým typem je samotné *JSX*, které tvoří komponentu. Druhým typem jsou právě komponenty funkcionální, které navracejí rovnou *JSX*, specifikující danou komponentu.

CSS styl jsem za pomoci knihovny *styled-components* deklaroval již v rámci samotného *TypeScript* souboru. Takto vytvořené komponenty jsem dále využil pro samotné zobrazení.

```
const Helpline = styled.div‘
    text-align: center;
    margin-top: 2rem;
    margin-bottom: 5rem;
‘
export default () => <Helpline>Need help with your tickets?</Helpline>
```

Výpis kódu 5.4: Příklad implementace funkcionální komponenty za využití knihovny `styled-components`



(a) Desktop (b) Mobilní zařízení

Obrázek 5.4: Wireframe komponenty zobrazující lístek pro desktop a mobilní zařízení

Kapitola 6

Zhodnocení praxe

Bakalářskou práci formou praxe jsem si vybral především z důvodu otestování mých dosavadních znalostí při reálném využití. Objektivně musím uznat, že praxe pro mě měla, z pohledu mých znalostí před nástupem na praxi, obrovský přínos. Během praxe jsem měl možnost si vyzkoušet příkladnou týmovou spolupráci a práci pod vedením, stejně jako po boku zkušených vývojářů. Tato zkušenost byla přínosná nejen z profesního hlediska, ale také hlediska osobního.

Z pohledu nabytých teoretických znalostí považuji za nejvíce přínosné seznámení s komplexnějším návrhem informačních systému. Získaný přehled o návrhových vzorech a možnost o nich diskutovat s kolegy, či možnost účastnit se návrhu a analýzy jednotlivých zadání jen zesílily můj zájem v této analytické části vývoje software.

Z technického pohledu považuji za stěžejní rozhodnutí, které platformě bych se rád v dalším uplatnění věnoval. Zkušenosti a znalosti získané v rámci platformy .NET a cloudové platformy *Azure* posunuly mé budoucí zaměření právě k technologiím společnosti Microsoft.

6.1 Stav projektu

V době psaní této práce je projekt London Theatre Direct ETickets již v produkčním provozu. Zákazník je po úspěšné objednávce přesměrován na prezentační web [B.2], kde jsou mu na vyžádání vygenerovány digitální lístky [A.1].

Během mé praxe ve společnosti ICT Capital byl do výsledného řešení zakomponován požadavek na integraci řešení generující digitální lístky ve formě *pdf* souborů. Na implementaci tohoto řešení jsem se aktivně podílel zejména na konzultační úrovni díky mé znalosti projektu. Implementací *pdf* lístků se z této služby stal primární zdroj distribuovaných divadelních vstupenek v rámci projektu London Theatre Direct.

6.2 Uplatněné znalosti získané během bakalářského studia

Při plnění úkolů zadaných při bakalářské praxi jsem pracoval výhradně s technologiemi založenými na platformě .NET. Z technologického hlediska byly tedy pro mě stěžejní znalosti z předmětů *Programovací Jazyky II* a *Architektura Technologie .NET*. Tyto technické znalosti doplněné znalostmi SQL databází, zejména z předmětu *Databázové a informační systémy* a základy návrhu software z předmětů *Úvod do softwarového inženýrství* a *Vývoj informačních systémů*, položily základ mému úspěšnému působení ve firmě ICT Capital.

Při vývoji webové prezentace jsem nejvíce ocenil znalosti z předmětů *Tvorba aplikací pro mobilní zařízení I* a *Vývoj internetových aplikací*, na které jsem navázal při studiu *frontend* frameworku *ReactJS* a obecně implementace webových prezentací.

6.3 Scházející znalosti získané na praxi

Jak je již patrné z předchozí podkapitoly, na poli *frontend* vývoje jsem před nástupem na praxi měl jen základní znalosti. Díky bakalářskému projektu jsem se naučil komplexněji využívat jazyk *JavaScript* a jeho nadstavbu *TypeScript*. Tyto nově nabyté znalosti mě výrazně posunuly v oblasti *frontend* vývoje a využití frameworku *ReactJS*.

Nemalé úsilí vyžadovalo také rozšíření znalostí v rámci platformy .NET, a to zejména při implementaci a integraci knihovny *MediatR* (strana 32), či využití frameworku Entity Framework Core.

Značně úzký rozhled jsem začal pociťovat také v rámci návrhu software, kde jsem hned po nástupu na praxi musel dohánět nedostatky na tomto velice důležitém poli při vývoji software. Tyto nedostatky se projevily zejména při implementace produkční služby [5.2], kde bych byl schopen rychleji detekovat nevhodnost implementovaného návrhového vzoru a popř. navrhnout alternativu.

Kapitola 7

Závěr

Náplní mé bakalářské praxe byla implementace mikroslužby zastřešující distribuci digitálních vstupenek na divadelní představení. Díky tohoto projektu jsem měl možnost spolupracovat na rozsáhlém přepisu stávajícího řešení, a to nejen z hlediska samotného kódu, ale zejména celkové architektury tohoto řešení. Práce na tomto projektu byla náročná nejen svými implementačními aspekty, kdy jsem implementoval a integroval dva kompletně rozdílné typy externích služeb spolu s webovou prezentací, ale zejména aspekty každodenního komerčního vývoje software.

Práce v týmu zkušených vývojářů a pro zahraničního zákazníka je pro studenta informačních technologií velkou výzvou. Díky této výzvě, poskytnuté společností ICT Capital, jsem měl možnost nejen rozšířit své znalosti a dovednosti v oboru informačních technologií, ale také se posunout z osobního hlediska. Vývoj v týmu a seznámení se s managementem projektu mě naučilo, že vývoj software je nejen o programování, ale také o komunikaci v rámci týmu a organizaci práce mi svěřené.

Při zápisu na bakalářskou praxi jsem zmíněnou výzvu přijímal s tím, že mi pomůže modelovat mé budoucí zaměření v tomto oboru. Při psaní této závěrečné kapitoly musím konstatovat, že tato praxe a zejména projekt, na kterém jsem mohl pracovat, tento cíl splnily.

Literatura

1. *ITIXO* [online] [cit. 2021-04-12]. Dostupné z: https://www.itixo.com/cs.
2. MICROSOFT. *.NET* [online] [cit. 2021-04-12]. Dostupné z: https://dotnet.microsoft.com/.
3. *Riganti* [online] [cit. 2021-04-12]. Dostupné z: https://www.riganti.cz/.
4. *Update Conference* [online] [cit. 2021-04-12]. Dostupné z: https://www.updateconference.net/cs.
5. RICHARDSON, Chris. *Pattern: Monolithic Architecture?* [Online] [cit. 2021-04-12]. Dostupné z: https://microservices.io/patterns/monolithic.html.
6. FOWLER, Martin; RICE, David; FOEMMEL, Matthew; HIEATT, Edward; MEE, Robert; STAFFORD, Randy. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
7. RICHARDSON, Chris. *What are microservices?* [Online] [cit. 2021-04-12]. Dostupné z: https://microservices.io/.
8. NEWMAN, Sam. *Monolith to Microservices*. O'Reilly Media, Inc, 2019.
9. APPLE. *Apple Wallet Developer Guide* [online]. 2018-01-16 [cit. 2021-04-12]. Dostupné z: https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/PassKit_PG/.
10. GOOGLE. *Google Pay for Passes* [online] [cit. 2021-04-12]. Dostupné z: https://developers.google.com/pay/passes.
11. MCGUINNESS, Tomas. *dotnet-passbook* [online] [cit. 2021-04-12]. Dostupné z: https://github.com/tomasmcguinness/dotnet-passbook.
12. EVANS, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
13. MICROSOFT. *CQRS Pattern* [online]. 2020-02-11 [cit. 2021-04-12]. Dostupné z: https://docs.microsoft.com/cs-cz/azure/architecture/patterns/cqrs.

14. BOGARD, Jimmy. *MediatR* [online] [cit. 2021-04-12]. Dostupné z: `https://github.com/jbogard/MediatR`.

15. BOGARD, Jimmy. *AutoMapper Documentation* [online] [cit. 2021-04-12]. Dostupné z: `https://docs.automapper.org/en/latest/index.html`.

16. FACEBOOK. *ReactJS* [online] [cit. 2021-04-12]. Dostupné z: `https://reactjs.org/`.

17. *TypeScript* [online] [cit. 2021-04-12]. Dostupné z: `https://www.typescriptlang.org/`.

18. *styled-components Documentation* [online] [cit. 2021-04-12]. Dostupné z: `https://styled-components.com/docs`.

Příloha A

Vzhled vygenerovaných digitálních lístků

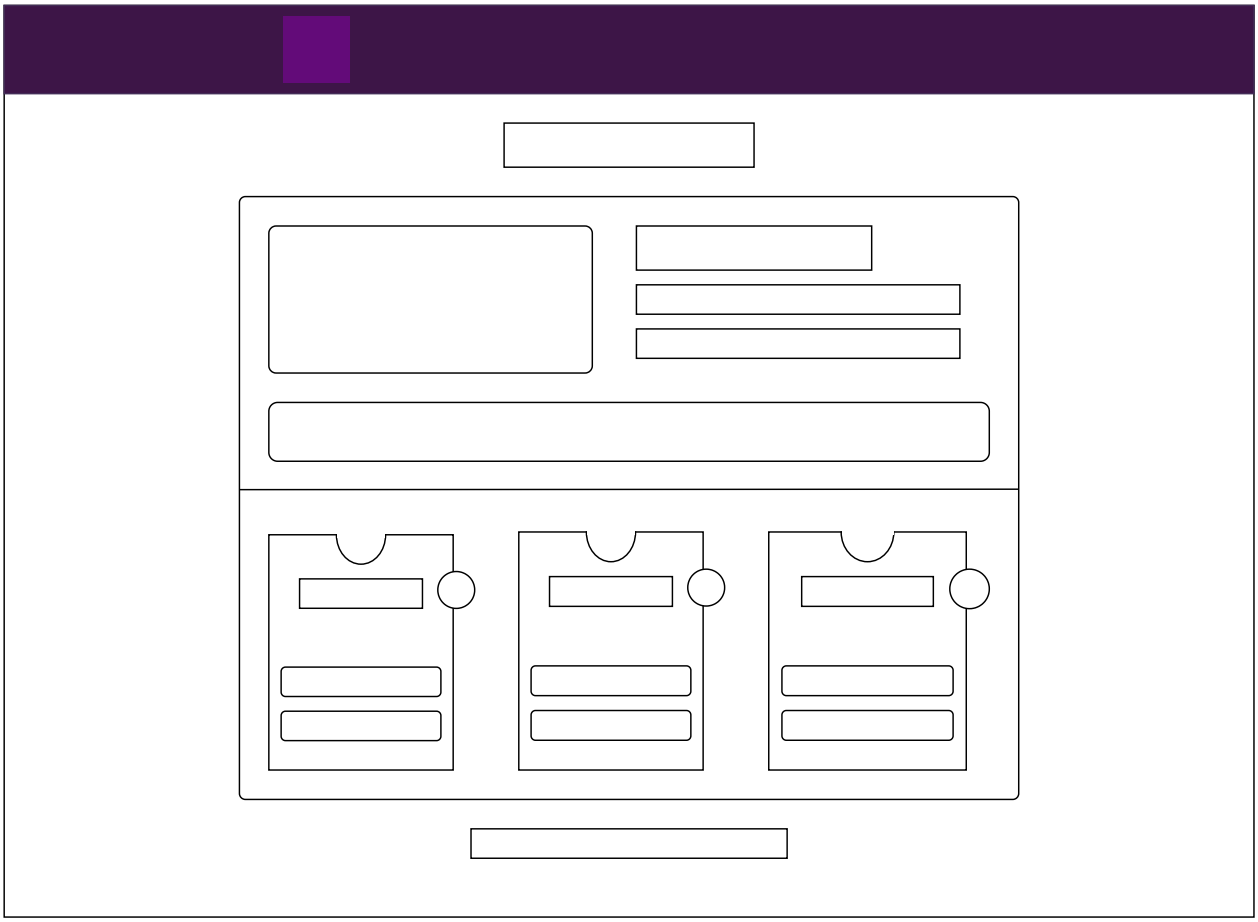
(a) Apple Wallet lístek

(b) Google Pay lístek

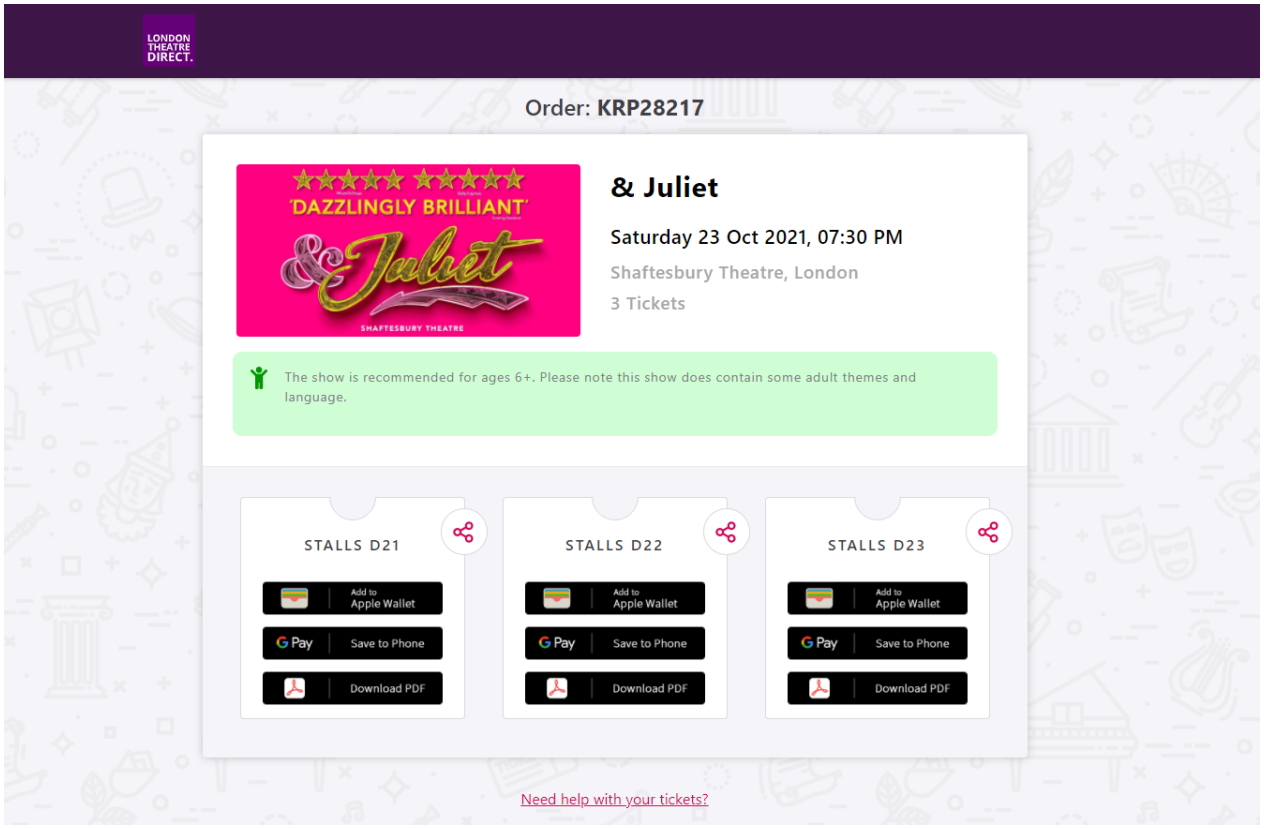
Obrázek A.1: Vygenerované lístky pro integrace *Apple Wallet* a *Google Pay*

Příloha B

Webová prezentace projektu London Theatre Direct ETickets



Obrázek B.1: Wireframe webové prezentace služby London Theatre Direct ETickets



Obrázek B.2: Výsledný vzhled prezentačního webu